Massive Model Rendering Techniques

Andreas Dietrich[1] Enrico Gobbetti[2] Sung-Eui Yoon[3]

[1]Computer Graphics Group
Saarland University
dietrich@cs.uni-sb.de

[2]Visual Computing Group
CRS4
gobbetti@crs4.it

[3]Department of Computer Science
KAIST
sungeui@cs.kaist.ac.kr

***Abstract*— We present an overview of current real-time massive model visualization technology, with the goal of providing readers with a high level understanding of the domain, as well as with pointers to the literature.**

***Index Terms*— real-time rendering, massive model visualization.**

I. Introduction

Interactive visualization and exploration of massive 3D models is a crucial component of many scientific and engineering disciplines and is becoming increasingly important for simulations, education, and entertainment applications such as movies and games. In all those fields, we are observing *data explosion*, i.e., information quantity is exponentially increasing. Typical sources of rapidly increasing massive data include the following:

- **Large-scale engineering projects.** Today, complete aircrafts, ships, cars, etc. are designed purely digital. Usually, many geographically dispersed teams are involved in such a complex process, creating thousands of different parts that are modeled at the highest possibly accuracy. For example, the Boeing 777 airplane seen in Figure 1a consists of more than 13,000 individual parts.
- **Scientific simulations.** Numerical simulations of natural real world effects can produce vast amounts of data that need to visualized to be scientifically interpreted. Examples include nuclear reactions, jet engine combustion, and fluid-dynamics to mention a few. Increased numerical accuracy as well as faster computation can lead to datasets of gigabyte or even terabyte size (Figure 1b).
- **Acquisition and measuring of real-world objects.** Apart from modeling and computing geometry, scanning of real-world objects is a common way of acquiring model data. Improvements in measuring equipment allows scanning in sub-mm accuracy range, which can result in millions to billions of samples per object (Figure 1c).
- **Modeling natural environments.** Natural landscapes contain an incredible amount of visual detail. Even for a limited field of view, hundreds of thousands of individual plants might be visible. Moreover, plants are made of highly complex structures themselves, e.g., countless leaves, complicated branchings, wrinkled bark, etc. Even modeling only some of these effects can produce excessive quantities of data. For example, the landscape model depicted in Figure 1d measures "only" a square area of 82 km $\times$ 82 km.

Handling such massive models presents important challenges to developers. This is particularly true for highly interactive 3D programs, such as visual simulations and virtual environments, with their inherent focus on interactive, low-latency, and real-time processing.

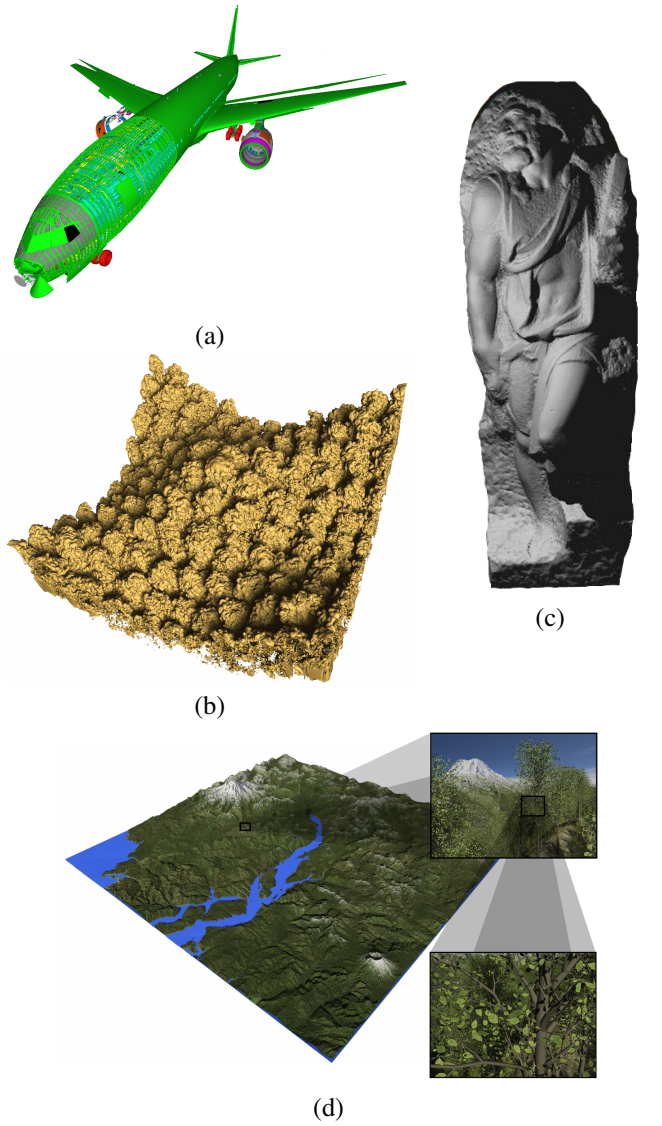


Fig. 1. **Examples of highly complex scenes.** (a) A Boeing 777 CAD model, made of more than 350 million triangles (12 GByte). (b) One time-step of a "Richtmyer-Meshkov" instability of turbulent fluids. The volumetric representation is described by 2048 $\times$ 2048 $\times$ 1970 samples. All 270 time-steps require 1.5 TByte. (c) A scanned model of Michelangelo's St. Matthew statue. The scene is represented by 372 million triangles (9.6 GByte). (d) A model of the Puget Sound Area, where an elevation map based on satellite data (20 GByte) has been populated with instantiated tree models, resulting in over 90 trillion potentially visible triangles.

In the last decade, the graphics community has witnessed tremendous improvements in the performance and capabilities of computing and graphics hardware. It therefore naturally arises the question if such a performance boost does not transform rendering performance problems into memories of the past. A single standard dual-core 3 GHz Opteron processor has roughly 20 GFlops, a Playstation 3's CELL processor has 180 GFlops, and recent GPUs, now fully programmable, provide around 340 GFlops. With the increased application of hardware parallelism, e.g., in the form of multi-core CPUs or multi-pipe GPUs, the performance improvements, which tend to follow, and even outpace, Gordon Moore's exponential growth prediction, seem to be continuing for a near future to come. For instance, Intel has already announced an 80 core processor capable of TeraFlop performance. Despite such an observed and continuing increase in computing and graphics processing power, it is however clear to the graphics community that one cannot just rely on hardware developments to cope with any data size within the foreseeable future. This is not only because the increased computing power also allows users to produce more and more complex datasets, but also because memory bandwidth grows at a significantly slower rate than processing power and becomes the major bottleneck when dealing with massive datasets.

As a result, massive datasets cannot be interactively rendered by brute force methods. To overcome this limitation, researchers have proposed a wide variety of output-sensitive rendering algorithms, i.e., rendering techniques whose runtime and memory footprint is proportional to the number of image pixels, not to the total model complexity. In addition to requiring out-of-core data management, for handling datasets larger than main memory or for providing applications the ability to explore data stored on remote servers, these methods require the integration of techniques for filtering out as efficiently as possible the data that is not contributing to a particular image.

This article provides an overview of current massive model rendering technology, with the goal of providing readers with a high level understanding of the domain, as well as with pointers to the literature. The main focus will be on rendering of large static polygonal models, which are by far the current main test case for massive model visualization. We will first discuss the two main rendering techniques (Section II) employed in rendering massive models: rasterization and ray tracing. We will then illustrate how rendering complexity can be reduced by employing appropriate data structures and algorithms for visibility or detail culling, as well as by choosing alternate graphics primitive representations (Section III). We will further focus on data management (Section IV) and parallel processing issues (Section V), which are increasingly important on current architectures. The article concludes with an overview of how the various techniques are integrated into representative state-of-the-art systems (Section VI), and a discussion of the benefits and limitations of the various approaches (Section VII).

II. Two Main Rendering Techniques

Rendering – the image generation process that takes places during visualization of geometric models – requires calculating each primitive's contribution to each pixel. It involves three main aspects: *projection*, *visible-surface determination*, and *shading*.

Projection transforms 3D objects into 2D for viewing. This is typically a planar perspective projection, where 3D points are projected onto a 2D image plane based on a center of projection (see Figure 3). Visible surface determination is necessary to compute which parts of a scene are actually visible by an observer, and which parts are occluded by other surfaces. Finally, shading means the computation of the appearance of visible surface fragments. For example, in case of photo-realistic image generation, shading can result in complicated light transport simulations, which in turn make it necessary to also determine visibility between surfaces.

As of today, practically only two rendering algorithms are almost exclusively applied: rasterization and ray tracing, which is mainly due to their robustness and simplicity.

A. Rasterization

Rasterization algorithms combined with the *Z-buffer* are widely used in interactive rendering, and are implemented in virtually all modern graphics boards in the form of highly parallel *graphics processing units* (GPUs). Rasterization is an example of an object-order rendering approach (also called forward-mapping): Objects to be rendered are sequentially projected onto the image plane, where they are rasterized into pixels and shaded. Visibility is resolved with the help of the Z-buffer, which stores for each pixel the distance of the respective visible object fragment to the observer.

This process can be efficiently realized in a pipeline setup, commonly known as *graphics pipeline*. Figure 2 illustrates a generic graphics pipeline architecture. Early graphics hardware was based on a hardwired implementation of this architecture, with multiple geometry and rasterization units made to work in parallel to further increase performance. In recent years, graphics hardware has started to feature extensions to the fixed-function pipeline, in a way that parts of the geometry stage and rasterizer stage can be programmed using vertex- and fragment-shaders. GPU designs have further evolved, and, nowadays, instead of being based on separate custom processors for vertex shaders, geometry shaders, and pixel shaders, the pipeline is realized on top of a large grid of data-parallel floating-point processors general enough to implement shader functionalities. Vertices, triangles, and pixels thus recirculate through the grid rather than flowing

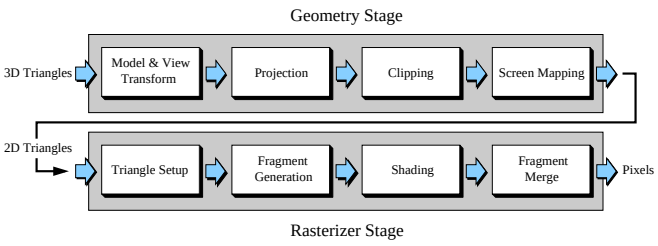


Fig. 2. **Generic rasterization graphics pipeline.** The pipeline consists of two main parts, geometry stage and rasterizer stage. Triangles are transformed into screen space, where they are rasterized and shaded.

through a pipeline with stages of fixed width, and allocation of the pool of processors to each shader type can dynamically vary to respond to varying graphics load.

A rasterization pipeline allows for processing arbitrary numbers of primitives in a stream-like manner, which is especially useful if scenes are rendered that do not fully fit into graphics or main memory. In this basic form rasterization techniques are limited to linear time complexity in the number of scene primitives, which is a direct result from the employed object order scheme. In order to enable rendering in logarithmic time complexity, spatial index structures need to be applied that allow for a-priori limiting the number of polygons to be sent down the graphics pipeline. Moreover, since the gap between GPU performance and bandwidth throughout the memory hierarchy is growing, appropriate techniques must be employed to carefully manage working set size and ensure coherent access patterns. We will see how this works in the following sections.

B. Ray Tracing

In contrast to rasterization, *ray casting* and its recursive extension *ray tracing* are image order rendering (backward mapping) approaches. Ray tracing closely models physical light transport as straight lines. In their classical form, ray tracing algorithms start by shooting imaginary (primary) rays from the observer (i.e., the projection center) through the pixel grid into a 3D scene (see Figure 3). For each ray the closest intersection with the model's surfaces is determined. To find out if a surface hitpoint is lit, so-called shadow rays are spawned into the direction of each light source. If such a ray is blocked, the hitpoint lies in shadow. Light propagation between surfaces can be computed by recursively tracing secondary rays, which emanate from previously found hitpoints. In Figure 3, e.g., a reflective surface has been hit, thus, the incoming ray is mirrored and fired again to find surfaces that are visible in the reflection.

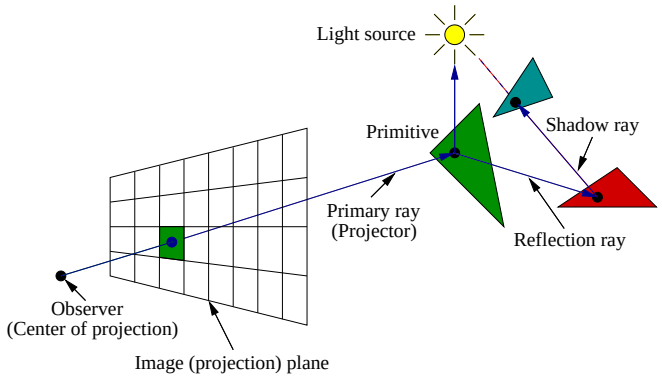


Fig. 3. **Recursive ray tracing.** A primary ray is fired from the observer through the pixel grid into a 3D scene. From the closest intersection point with the scene geometry secondary rays are spawned. This process can be continued recursively, e.g., in case of multiple reflections.

A basic ray tracing implementation can be very simplistic, and can be realized with much less effort than a (software) rasterizer. For example, all parts of the rasterizer geometry stage (see Figure 2) are handled implicitly as result of the backward projection property.

In a naive implementation, the closest intersection would be calculated by sequentially testing each primitive of the scene for each ray, which is obviously only practical for very small models. To limit the number of primitives for which the actual ray–primitive intersection test is performed, again spatial index structures are necessary. In a modern ray tracer such acceleration structures are typically considered to be an integral part of the algorithm, and allow for an average logarithmic time complexity with respect to the number of primitives.

Probably, the most well-known property of ray tracing is the ability to to generate highly photo-realistic imagery (an example can be seen in Figure 4). While in classical ray tracing rays are only shot into the most significant directions (e.g., towards lights), accurately computing global lighting (including glossiness and indirect light propagation) requires the solution of high-dimensional integrals over the path-space of light transport in a scene. Today, numeric *Monte-Carlo* (MC) integration techniques in combination with ray tracing are used almost exclusively for this purpose [1].

Fig. 4. **Global illumination.** A photo-realistic ray traced image of the Sponza Atrium, rendered using global illumination Monte Carlo techniques. Note the indirect lighting of the hallway.

Another advantage resulting from the recursive nature of ray tracing algorithms is that shaders can be deployed in a plug-and-play manner. The characteristics of different surfaces can be described independently, and all optical effects are automatically combined in a physically correct way. The most eminent disadvantage of ray tracing is its high computational complexity, especially for higher order lighting effects. Because of this, it has been employed in a real-time context only in recent years [2]. While prototype hardware implementations exist [3], only software ray tracing has so far been applied to massive models.

An important ingredient that is widely applied in state-of-the-art real-time ray tracing systems is to simultaneously trace bundles of rays called *packets*. First, working on packets allows for using SIMD (single instruction multiple data) vector

operations of modern CPUs to perform parallel traversal and intersection of multiple rays. Second, packets enable deferred shading, i.e., it is not necessary to switch between intersection and shading routines for every single ray, thus amortizing memory accesses, function calls, etc. Third, frustum traversal methods [4] avoid traversal steps and intersection calculations based on the bounds of ray packets, therefore making better use of object as well as scanline coherence.

C. Summary

Both rasterization and ray tracing algorithms can perform rendering – and most importantly, visible-surface determination – in logarithmic time complexity with respect to scene size. However, this is only possible if spatial index structures and hierarchical rendering are involved (see below).

The main advantage of rasterization algorithms is the ability to efficiently exploit scanline coherence. Only the corners of triangles need to be explicitely projected, while the geometric properties of the pixels enclosed by the triangle edges can be easily interpolated during the actual rasterization step. Consequently, such techniques work best in cases where large screen space areas are covered by a few triangles. Conversely, ray tracing and ray casting perform better if visibility needs to be evaluated point-wise. Interestingly, hierarchical front-to-back rasterizing in combination with occlusion culling techniques can be interpreted as a form of beam- or frustum tracing.

When it comes to advanced shading effects, ray tracing algorithms are usually the premier choice, mainly because of their physical accuracy and implementation flexibility. Especially in highly complex models, the ability to perform shading and light transport adaptively, proves to be helpful. Implementing advanced global shading algorithms in rasterization architectures is more complex, because during the shading stage no global access to the scene database is possible, and visibility between surface points cannot explicitely be determined. The problem can in principle be overcome by computing maps, and applying them in successive rendering passes. However, this does not easily allow for adaptive rendering, e.g., controlling the reflection recursion depth on a per pixel basis.

As of today, most massive model rendering systems use exclusively one of the two presented rendering techniques. It is, however, likely that future systems will incorporate hybrid approaches, in particular as graphics hardware is becoming more and more general purpose oriented, and will allow for executing rasterization and ray tracing side-by-side.

III. Complexity Reduction Techniques

In order to meet timing constraints, massive model visualization applications have to employ methods for filtering out as efficiently as possible the data that is not contributing to a particular image.

One straightforward approach is to simplify complex models until they become manageable by the application: if models are too complex, make them simpler! This static "throw-away-input-data approach" might seem simplistic, but can be considered beneficial in a number of practical use cases. A common application of static simplification is reducing the complexity of very densely over-sampled models. For instance, models generated by scanning devices and iso-surfaces extracted by algorithms such as marching cubes are often uniformly over-tessellated because of the nature of most reconstruction algorithms. By adaptively simplifying meshes so that local triangle density adapts to local curvature it is often possible to radically reduce triangles without noticeable error. More generally, users may want to produce an approximation which is tailored for a specific use, e.g., viewing from a distance. In the more general case, however, the quality loss incurred when using off-line simplification techniques is not acceptable, and the application must resort to more general adaptive techniques able to filter data at run-time. These methods can be broadly classified into view-dependent level-of-detail algorithms and visibility culling methods. Level-of-detail (LOD) techniques reduce memory access and rendering complexity by exploiting multi-resolution data structures for dynamically adapting the resolution of the dataset (the number of required model samples per pixel), while visibility culling techniques achieve the same goal by avoiding the processing of parts that can be proved invisible because out of view (in the case of view-frustum culling) or masked (in the case of occlusion culling).

A. Geometric Simplification

Geometric simplification is a well studied subject, and a number of high quality automatic simplification techniques have been developed [5].

The wide majority of the methods iteratively simplifies an input mesh by sequences of vertex removal or edge contraction (see Figure 5). In the first case of Figure 5, originally introduced by Schroeder [6], at each simplification step, a vertex is removed from the mesh and the resulting hole is triangulated. In the second case, popularized by Hoppe [7], the two endpoints of an edge are contracted to a single point and the triangles that degenerate in the process are removed.

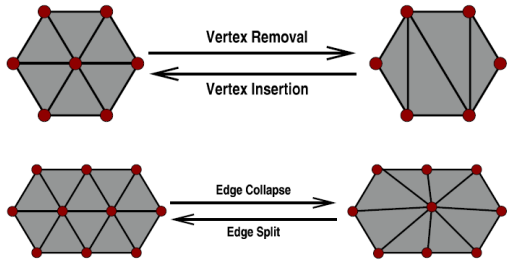


Fig. 5. **Mesh simplification primitives and their inverses.** Top: A vertex is removed and the resulting hole triangulated. Bottom: An edge is collapsed to a single point.

The control of the *approximation accuracy* is critical in the process, if only to guide the order in which to perform operations. The error evaluation method most frequently used in current applications is the *quadric error metrics* (QEM), originally proposed by Garland and Heckbert in [8], in which a quadric (represented by a symmetric, positive semi-definite matrix) is associated with each vertex, subsuming the sum of the squared distances of the vertex to all its incident

planes. Using quadrics has the major benefit of reducing space overheads to storing a small symmetric matrix per vertex, and leads to an extremely fast error metric consisting of simple vector–matrix operations.

In most current systems, simplification is performed in an iterative greedy fashion, which maintains a sorted list of candidate operations and applies at each step the operation associated to the minimal simplification error. Many variations of this approach have been proposed, especially for dealing with extremely large meshes not fitting in main memory. An efficient approach recently introduced in this context is the idea of "streaming simplification" [9]. The key insight is to keep input and output data in streams and document, for example, when all triangles around a vertex or all points in a particular spatial region have arrived with "finalization tags". This representation allows for streaming very large meshes through main memory while maintaining information about the visiting status of edges and vertices. At any time, only a small portion of the mesh is kept in-core, with the bulk of the mesh data residing on disk. Mesh access is restricted to a fixed traversal order, but full connectivity and geometry information is available for the active elements of the traversal. For simplification, an in-core buffer is filled and simplified and output is generated as soon as enough data is available.

Geometric simplification can be considered a mature field for which industrial quality solution exists. However, these methods, that repeatedly merge nearby surface points or mesh vertices based on error minimization considerations, perform best for highly tessellated surfaces that are otherwise relatively smooth and topologically simple. However, it becomes difficult, in other cases, to derive good "average" merged properties. Geometric simplification is thus hard to apply when the visual appearance of an object depends on resolving the ordering and mutual occlusion of even very close-by surfaces, potentially with different shading properties.

B. Level-of-Detail

A *level-of-detail* (LOD) model is a compact description of multiple representations of a single shape and is the key element for providing the necessary degrees of freedom to achieve run-time adaptivity. LOD models can be classified as *discrete*, *progressive*, and *continuous* LOD models.

Discrete LOD models simply consist of ordered sequences of representations of a shape, representing an entity at increasing resolution and accuracy. The expressive power of discrete LODs is limited to the different models contained in the sequence: these are a small number and their accuracy/resolution is predefined (in general, it is uniform in space). Thus, the possibility of adapting to the needs of user applications is scarce. The extraction of a mesh at a given accuracy reduces to selecting the corresponding mesh in the sequence, whose characteristics are the closest to application needs. Such models are standard technology in graphics languages and packages, such as VRML or X3D and are used to improve efficiency of rendering: depending on the distance from the observer, or a similar measure, one of the available models is selected. The approach works well for small or distant isolated objects, which can be found in CAD models [10]. However, it is not efficient for large objects spanning a range of different distances from the observer. Since there is no relation among the different LODs, there are no constraints on how the various detail models are constructed.

Progressive models consist of a coarse shape representation and of a sequence of small modifications which, when applied to the coarse representation, produce representations at intermediate levels of detail. Such models lead to very compact data structures, based on the fact that all modifications in the sequence belong to a predefined type, and thus can be described with a few parameters. The most notable example is the Progressive Mesh representation [7], included in the DirectX library since version 8. In this case, the coarsening/refinement operations are edge collapse/edge split. A mesh at uniform accuracy can be extracted by starting from the initial one, scanning the list of modifications and performing modifications from the sequence until the desired accuracy is obtained. As for discrete LODs, the approach works well for small or distant isolated objects, but it is not efficient for large objects spanning a range of different distances from the observer.

Continuous LOD models improve over progressive models by fully supporting selective refinement, i.e., the extraction of representations with an LOD that can be variable in different parts of the representation, and can be changed on a virtually continuous scale. Continuous LODs are typically created using a refinement/coarsening process similar to the one employed in progressive models. However, rather than just storing a totally ordered sequence of local modifications, continuous LOD models link each local modification to the set of modifications that block it. Thus, contrary to progressive models, local updates can be performed without complicated procedures to find out dependency between modifications. A general framework for managing continuous LOD models is the multi-triangulation [11], which is based on the idea of encoding the partial order describing mutual dependencies between updates as a directed acyclic graph (DAG), where nodes represent mesh updates (removals and insertions of triangles), and arcs represent relations among updates. An arc $a = (n_1, n_2)$ exists if a non-empty subset of the triangles introduced by n_1 are removed by n_2. Selectively refined meshes can thus be obtained from cuts of this graph, and by sweeping the cut forward/backward through the DAG the resolution increases/decreases. Figure 6 illustrates the concept with an an example.

Most of the continuous LOD models can be expressed in this framework. Many variations have been proposed. Up until recently, however, the vast majority of view-dependent level-of-detail methods were all based on multi-resolution structures taking decisions at the triangle/vertex primitive level. This kind of approach involves a constant CPU workload for each triangle that makes detail selection the bottleneck of the whole rendering process. This problem is particularly stringent in rasterization approaches, because of the increasing CPU/GPU performance gap. To overcome this bottleneck and to fully exploit the capabilities of current hardware it is therefore necessary to select and send batches of geometric primitives

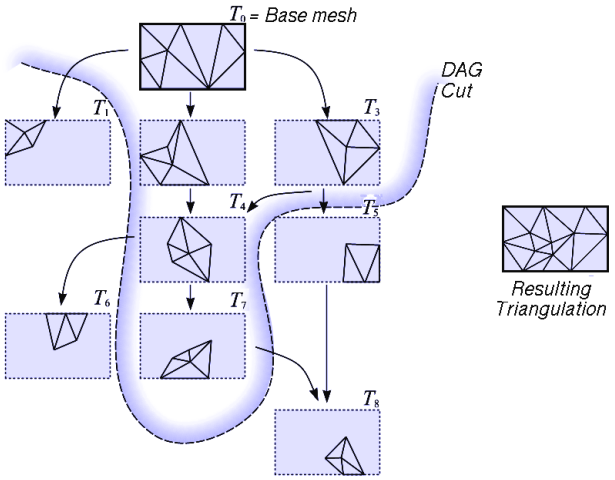


Fig. 6. **Multi-triangulation.** A sequence of local modification over a mesh is coded as DAG over the fragments T_i; a cut of the DAG defines a conforming triangulation that can obtained by pasting all the fragments above the cut.

to be rendered with just a few CPU instructions. To this end, various GPU oriented multi-resolution structures have been recently proposed, based on the idea of moving the granularity of the representation from triangles to triangle patches [12], [13]. Thus, instead of working directly at the triangle level, the models are first partitioned into blocks containing many triangles, and, then, a multi-resolution structure is constructed among partitions. By carefully choosing appropriate subdivision structures for the partitioning and managing boundary constraints, hole-free adaptive models can be constructed.

The benefit of these approaches is that the needed per-triangle workload to extract a multi-resolution model reduces by orders of magnitude. The small patches can be preprocessed and optimized off line for a more efficient rendering, and highly efficient retained mode graphics calls can be exploited for caching the current adaptive model in video memory. Recent work has shown that the vast performance increase in CPU/GPU communication results in greatly improved frame rates [12], [13].

C. Visibility Culling

Often, massive scenes are densely occluded or are too large to be viewed in their entirety from a single viewpoint, which means that in most viewing situations only a fraction of the model geometry is actually visible. For instance, in the Boeing 777 model in Figure 1a this can be seen in the rear section, where the hull completely covers the internal airplane structure. Therefore, a straightforward strategy would be to determine the *visible set*, i.e., the set of objects that contribute to the current image. The intention is to reject large parts of the scene before the actual visible surface determination takes place, thereby reducing the rendering complexity to the complexity of the computed subset of the scene geometry. This process of computing a visible subset of a scene is termed *visibility culling* [14], and is the other essential ingredient, in addition to LOD techniques, to make applications output-sensitive.

Three typical culling examples are *back-face culling*, *view-frustum culling*, and *occlusion culling* (see Figure 7). Back-face and view-frustum culling are trivial to implement, as they are local per-primitive operations. Occlusion culling is a far more effective technique since it removes primitives that are blocked by groups of other objects, but is unfortunately not as trivial to handle as the first two culling techniques because of its global nature. Often preprocessing needs to be involved, usually to generate some sort of scene hierarchies to allow for performing occlusion test in a top-down order.

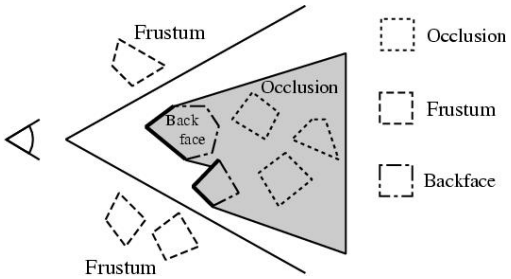


Fig. 7. **Visibility culling.** Back-face culling, view-frustum culling, and occlusion culling are the most typical examples of methods for computing a visible subset of a scene.

Quite a lot of different occlusion strategies have been proposed so far. Occlusion culling approaches are broadly classified into *from-point* and *from-region* visibility algorithms [14]. From-region algorithms compute a *potentially visible set* (PVS) for cells of a fixed subdivision of the scene and are computed offline in a preprocessing phase. During rendering only the primitives in the PVS of the cell where the observer is currently located, are rendered. From-region methods are mainly used for specialized applications, e.g., like visualization of urban scenarios or building interiors. From-point algorithms, on the other hand, are applied online for each particular viewpoint, and are usually better suited for general scenes, since for general environments accurate PVSs are hard to compute.

Visibility culling methods are typically realized with the help of a so-called *spatial index*, a spatial data structure that organizes geometry in 3D space. There are two major approaches, *bounding volume hierarchies* (BVHs) and *spatial partitioning*.

Bounding volume hierarchies organize geometry bottom-up. Groups of objects are encapsulated by a larger volume, which encloses them completely. Then bounding volumes can again be grouped and put into even larger volumes. The resulting tree can be rendered in a top-down order, starting with the scene bounding box. If a bounding volume, i.e., its boundary, is found to be fully or partially visible, rendering continues with its child-volumes. If a volume is completely invisible, traversal of the respective sub-tree can be discontinued, since all children will not be visible either.

In contrast, spatial partitioning schemes subdivide the scene top-down. The scene bounding box is split into disjoint partitions, which may further be subdivided in the same fashion. Each atomic partition holds a list of primitives it contains in whole or in part, and which will be rasterized if the partition

can be classified as visible. In case of ray tracing, all primitives of such a partition are intersected sequentially with the current ray.

Quite a number of spatial partitioning schemes have been proposed in the past, most popular are hierarchical *grids*, *octrees*, *kd-trees*. More details can, e.g., be found in [15]. An example of a kd-tree scene partition is illustrated in Figure 8. Kd-trees are axis-aligned *binary space partitioning* (BSP) trees. Construction of a kd-tree starts with the bounding box of the model and a list of contained primitives. The scene bounding box is then subdivided into two sub-boxes along one of the three primary coordinate axes, and the list of primitives is sorted into the two halves, creating two primitive lists, one for each half. Polygons that lie in either half are simply replicated. The process is recursively continued for both sub-boxes and their respective primitive lists. This way a binary tree is constructed, where each node corresponds to a spatial region (called *voxel*), and its children to a binary space partition of their parent region. If splitting positions are chosen to tightly enclose the scene geometry, kd-trees typically exhibit superior culling efficiency over other acceleration structures.

When combined with ray tracing, each ray traverses the kd-tree top-down, resulting in a linear sequence of cells (leaf-voxels) the ray passes. In the example in Figure 8 the ray visits cells 3, 2, and 4. Primitives contained in these cells are tested sequentially, and traversal can be stopped if a hitpoint is found. Using a rasterizer, kd-tree traversal is performed similarly. However, here all cells are visited that intersect the viewing frustum. In our example this would be cells 3, 2, 4, and 5. Only the primitives of the respective cells are sent to the graphics pipeline.

In order to achieve a sub-linear time complexity, employing acceleration structures alone is not sufficient. It is also necessary to include an early traversal termination. For a ray tracer this is trivial since visibility is evaluated independently for each ray, and once a hitpoint has been found, it is certain that geometry behind is not visible.

Using rasterization, the decision whether traversal of the spatial index can be stopped can also be made in image space, by exploiting the Z-buffer, and the most recent algorithms exploit graphics hardware for this purpose. During rendering – when the spatial index is traversed hierarchically in a front-to-back order – the bounding box of each visited node is tested against the Z-buffer and traversal is aborted as soon as occlusion can be proved, i.e., when all Z-values of a box are behind the corresponding stored Z-buffer's values. An efficient implementation of this method requires the availability of rapid Z-queries for screen regions. A classic solution is the hierarchical Z-buffer, which extends the traditional Z-buffer to a hierarchical Z-pyramid maintaining for each coarser block the farthest Z-value among the corresponding finer level blocks, therefore allowing to quickly determine if a geometry is visible by a top-down visit of the Z-pyramid. A pure software implementation of this method is impractical, but to some extent this idea is exploited in the current generation of graphics hardware by applying early Z-tests of fragments in the graphics pipeline (e.g., ATI's Hyper-Z technology or NVIDIA's Z-cull), and providing users with so-called *occlusion queries*. These queries define a mechanism whereby an application can query the number of pixels (or, more precisely, samples) drawn by a primitive or group of primitives. For occlusion culling during scene traversal the faces of bounding boxes are simply tested for visibility against the current Z-buffer using the occlusion query functionality to determine whether to continue traversal. It should be noted that, although the query itself is processed quickly using the raw power GPU, its result is not available immediately due to the delay between issuing the query and its actual processing by the graphics pipeline. A naive application of occlusion queries can thus even decrease the overall application performance due the associated CPU stalls and GPU starvation. For this reason, modern methods exploit spatial and temporal coherence to schedule the issuing of queries [16], [13], [17]. The central idea of these method is to issue multiple queries for independent scene parts and to avoid repeated visibility tests of interior nodes by exploiting the coherence of visibility classification.

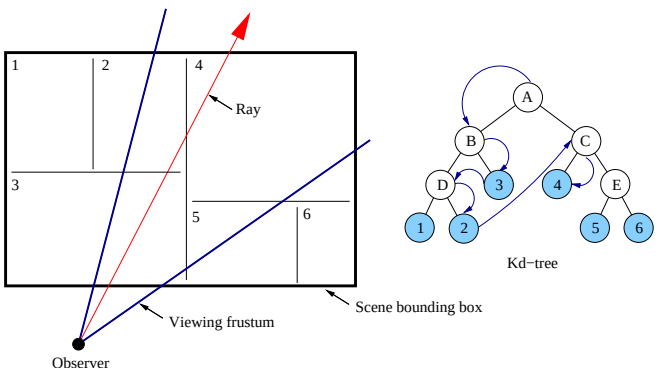


Fig. 8. **Kd-tree traversal.** Top-down traversal of the kd-tree results in a linear enumeration of the scene partitions a ray passes.

D. Summary

Level-of-detail and visibility culling techniques are fundamental ingredients for massive model rendering applications. It is important to note that, in general, the lack of one of these techniques limits the theoretical scalability of an application. However, massive models arise from a number of different domains, and the relative importance of the LOD management and visibility culling components depends on the widely varying geometry, appearance, and depth complexity characteristics of the models. For instance, typical 3D scanned models and terrain models tend to be extremely dense meshes with very low depth complexity, favoring pure LOD techniques, while architectural and engineering CAD models tend to combine complicated geometry and appearance with a large depth complexity, requiring applications to deal with visibility problems.

Few approaches exist that integrate LODs with occlusion culling both in the construction and rendering phases. Moreover, and most importantly, the off-line simplification process that generates the multi-resolution hierarchy from which view-dependent levels of detail are extracted at rendering time is essentially unaware of visibility. When approximating very

complex models, however, resolving the ordering and mutual occlusion of even very close-by surfaces, potentially with different shading properties, is of primary importance (see Figure 9). Providing good multi-scale visual approximations of general environments remains an open research problem, and the few solutions proposed so far involve the introduction of other primitives than triangle meshes for visibility-aware complexity reduction.

Representations other than polygons offer significant potential for massive models visualization. In conventional polygon-based computer graphics, models have become so complex that for most views polygons are smaller than one pixel in the final image. The benefits of polygons for interpolation and multi-scale surface representation become thus questionable. For these reasons, researchers have started investigating alternative approaches, that represents complex 3D environments with sets of points, voxels, or images. At present time, however, no single best representation exists in terms of storage, computational and implementation costs. For more information, see the sidebars "Alternative Geometric Representations" and "Image-Based Methods".

IV. Data Management Techniques

In the previous section, we have discussed complexity reduction techniques in order to improve the performance of rendering massive models. However, some of those techniques, e.g., LOD methods, can even increase the size of external data since we have to maintain different versions of a model. Moreover, we may have still a huge amount of in-core data, especially for creating high-resolution images, even after applying all those complexity reduction techniques.

Unfortunately, the computation trend during the last several decades is the aggravating performance of data access speed compared to that of processing speed [18]. Moreover, it is likely that this computational trend is to persist in the near future. Therefore, system architectures have been employing various caches and memory hierarchies to reduce expensive data access time and memory latency. Typically, the access times of different levels of a memory hierarchy vary by orders of magnitude (e.g., 10^{-8} s for L1/L2 caches, 10^{-7} s for main memory, and 10^{-2} s for disks). Also, as ubiquitous computing is more widely accepted, data is now accessed through the network in many cases, where the data access time is very expensive.

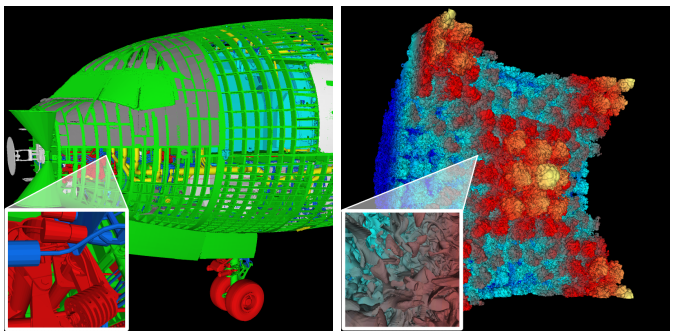


Fig. 9. **Boeing 777 engine details (left) and isosurface details (right).** These kinds of object, composed of many loosely connected interweaving detailed parts of complex topological structure, are very hard to simplify effectively using off-line geometric simplification methods that do not take into account visibility. As seen in the insets, a single pixel gets contributions from many mutually occluding colored surfaces.

As a result, it is critical to reduce the number of cache misses in order to maximally utilize the exponential growth rate of computational processing power and improve the performance of various applications including rasterization and ray tracing. In this section we will discuss three data management techniques: out-of-core techniques, layout methods, and compression methods, to improve the performance of applications by reducing data access time.

A. Out-of-Core Techniques

Out-of-core or external memory techniques store the major part of the scene database on disk, and only keep the fraction of the data that is currently processed in main memory. Such methods target to reduce the number of disk accesses, which are several orders of magnitude more expensive than that of memory and L1/L2 cache access time. In general, out-of-core techniques require two cache parameters: the size of available main memory and the disk block size. Since out-of-core techniques require these explicit cache parameters, they are also known as *cache-aware* techniques.

Given the known sizes of main memory and disk blocks, out-of-core techniques keep the working set size of rendering (or other applications) less than the size of main memory. They achieve this property typically by using an explicit data page system. Therefore, they avoid any I/O thrashing, where a huge number of disk cache misses occurs, and the performance of applications is severely degraded. Also, most out-of-core techniques construct compact external representations optimized towards disk block size to effectively reduce the number of disk cache misses. Also, these techniques are used together with pre-fetching methods to further reduce data access time. For readers interested further in out-of-core techniques, we refer to a survey of Silva et al. [19].

B. Layout Techniques

Most triangular meshes used in games and movies are typically embedded in two or three dimensional geometric space. However, these triangle meshes are stored on disk or in main memory as one dimensional linear data representations. Then, we access such data stored in the one dimensional data sequence using a virtual address or some other index format. An example of this mapping is shown in Figure 10. One implication of this mapping process of 3D or 2D triangular meshes into 1D linear layout representations is that two vertices (or triangles) close in the original 2D or 3D geometric space can be stored very far away in the 1D linear layout representation. This is mainly because a 1D layout cannot adequately represent the relationships of vertices and triangles embedded in 2D or 3D geometric space.

Many rendering methods including rasterization and ray tracing access vertices and triangles in a *coherent* manner. For example, suppose that a triangle and its three associated vertices are accessed for rendering. Then, it is most likely that

SIDEBAR: Alternative Geometric Representations

Multi-resolution hierarchies of point primitives have recently emerged as a viable alternative to the more traditional mesh refinement methods for interactively inspecting very large geometric models. One of the major benefits of this approach is its simplicity, stemming from the fact that there is no need to explicitly manage and maintain mesh connectivity during both preprocessing and rendering. The possibility of using points as a rendering primitives was first suggested by Levoy and Whitted [1], that pointed out that point primitives are more appropriate than triangles for complex, organic shapes with high geometric and appearance detail. Since then, a large body of work has been performed in the area of point based graphics.
The reconstruction of continuous (i.e., hole-free) images from a discrete set of surface samples is the major problem faced by point rendering approaches. It can be done by image-space reconstruction techniques [2] or by object-space resampling. The techniques from the latter category dynamically adjust the sampling rate so that the density of projected points meets the pixel resolution, which can be done both for rasterization and ray tracing approaches. Since this depends on the current viewing parameters, the resampling has to be done dynamically for each frame, and multi-resolution hierarchies or specialized procedural resampling techniques are exploited for this purpose. Examples are bounding sphere hierarchies [3], dynamic sampling of procedural geometries [4], the randomized Z-buffer [5], and the rendering of moving least squares (MLS) surfaces [6]. As for polygonal multi-resolution rendering, amortizing over a large number of primitives is essential to maximize rendering speed on current architectures, and the highest performance is currently obtained by coarse-grained approaches [7].
Overall, peak performance of high quality techniques is, however, currently inferior to the performance of corresponding triangle rasterization approaches, since current graphics hardware does not natively support essential point filtering and blending operations. This situation might change in the near future, as novel architectures for hardware-accelerated rendering primitives are currently being introduced [8].
Point based representations are appealing in massive model applications not only for rendering, but also to serve as modeling primitives for generating LODs. Classically, they have been used to represent surface elements. More recently, they have been used to model the appearance of small volumetric portions of the environment. In the Far Voxels approach [9], LODs are generated by discretizing spatial regions into cubical voxels. Each voxel contains a compact direction dependent approximation of the appearance of the associated volumetric subpart of the model when viewed from a distance. The approximation is constructed by a visibility aware algorithm that fits parametric shaders to samples obtained by casting rays against the full resolution dataset, and is rendered using a point primitives interpreted by GPU shaders. A similar approach to model simplification is also applicable to ray tracing [10], [11].

References

[1] Marc Levoy and Turner Whitted. The Use of Points as a Display Primitive. Technical Report TR 85-022, University of North Carolina at Chapel Hill, 1985.

[2] J.P. Grossman and William J. Dally. Point Sample Rendering. In *Rendering Techniques 1998 (Proceedings of the Eurographics Workshop on Rendering)*, pages 181–192, 1998.

[3] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 343–352, 2000.

[4] Marc Stamminger and George Drettakis. Interactive Sampling and Rendering for Complex and Procedural Geometry. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pages 151–162, 2001.

[5] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 361–370, 2001.

[6] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Cláudio T. Silva. Point Set Surfaces. In *Proceedings of IEEE Visualization 2001*, pages 21–28, 2001.

[7] Enrico Gobbetti and Fabio Marton. Layered Point Clouds: a Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-Sampled Models. *Computers & Graphics*, 28(6):815–826, 2004.

[8] Tim Weyrich, Cyril Flaig, Simon Heinzle, Simon Mall, Timo Aila, Kaspar Rohrer, Daniel Fasnacht, Norbert Felber, Stephan Oetiker, Hubert Kaeslin, Mario Botsch, and Markus Gross. A Hardware Architecture for Surface Splatting. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, page 90, 2007.

[9] Enrico Gobbetti and Fabio Marton. Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 878–885, 2005.

[10] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-LODs: Fast LOD-Based Ray Tracing of Massive Models. *The Visual Computer*, 22(9–11):772–784, 2006.

[11] Andreas Dietrich, Joerg Schmittler, and Philipp Slusallek. World-Space Sample Caching for Efficient Ray Tracing of Highly Complex Scenes. Technical Report TR-2006-01, Computer Graphics Group, Saarland University, 2006.

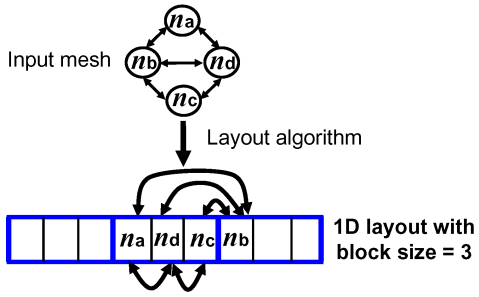


Fig. 10. **One dimensional layout.** This figure illustrates a layout computation method of an input mesh. Since many modern architectures using block-based caches assume a coherent access of runtime applications, the mesh should be stored in the layout in a coherent manner.

neighboring triangles or vertices will be accessed for rasterizing the next triangles or performing ray–triangle intersection tests for subsequent rays. Therefore, two vertices (or triangles) close in original geometric space are most likely to be accessed sequentially rather than any triangle or vertex will be equally likely to be accessed.

Although rendering applications access data in a coherent manner in the geometric space where triangle meshes are embedded, there is no guarantee that we will have coherent data access patterns in terms of 1D data layout representations due to the intrinsic lower dimensionality of 1D layouts compared to those of meshes. Particularly, this phenomenon can have significant impact on the performance of many applications running on modern computer architectures. This is mainly due to I/O mechanisms of most modern computer architectures.

Most I/O architectures use hierarchies of memory levels, where each level of memory serves as a *cache* for the next level. Memory hierarchies have two major characteristics. First, lower levels are larger in size and farther from the processor and, thus, have lower access times. Second, data is moved in large blocks between different memory levels. Data is initially stored in the lowest memory level, typically the disk or can be even accessed through the network. A transfer is performed whenever there is a cache miss between two adjacent levels of the memory hierarchy.

This *block fetching* mechanism assumes that runtime applications will access data in a coherent manner. As mentioned earlier, most rendering methods including rasterization and ray tracing access data in a coherent manner, only in the geometric space where meshes are embedded. However, data accessed in the 1D layout may be stored very far away and, thus, can require a lot of block fetching, i.e., cache misses, during runtime applications.

SIDEBAR: Image-Based Methods

In the geometry-based rendering approach, the visible component of the world is the union of two elements: the geometric description of the objects and the color and lighting conditions. A different approach is to consider the world as an infinite collection of images, one for each position, orientation and time. Such a collection of images would realize what is called *plenoptic function*, i.e., a function returning the color perceived from a specified eye position, view orientation and time. The goal of *image-based rendering* (IBR) is to generate images by resampling the plenoptic function given view parameters [1]. While in the general case a fully IBR approach is typically impractical, due to the sheer amount of data required for a full light field encoding of a scene, in the last decade a set of successful hybrid techniques have been proposed to accelerate the rendering of portions of a complex scene, with the replacement of complex geometry with textures in well defined particular cases. In most cases, the basic idea is to use a geometry-based approach for near objects, and switch to a radically different image-based representation, called *impostor*, for distant objects having a small, slowly changing on-screen projection.

The *billboard*, i.e., a textured planar polygon whose orientation changes to always face the viewer is possibly the most well-known image-based representation, and is used for replacing geometric representations of objects that have a rough cylindric symmetry, like a tree.

Another application of IBR is in environments which are naturally subdivided in cells with reduced mutual visibility. A typical example is the inside of a building, where adjacent rooms can be connected by doors of windows and if the observer is in a room he/she can see the inside of the adjacent cells only through those openings. This feature can be exploited in visibility culling, disregarding all the geometry which is outside the perspective formed by the observer position and the opening. If the observer is not too close to the opening and/or the opening is not too wide, it makes sense to put a texture on the opening instead of displaying the geometry. In [2] *portal textures* are introduced to this aim.

These simple approaches are limited by the fact that a single texture provides the correct view of the scene only from the point where it has been sampled and not elsewhere, leading to artifacts when the observer moves. For these reasons, a number of authors have proposed more elaborate solutions to incorporate parallax effects, such as *textured depth meshes* [3], in which textures are triangulated and a depth value is associated to each vertex, and *layered depth images* [4], that for each pixel store all the intersections of the view ray with the scene.

These techniques, introduced a decade ago are enjoying a renewed interest, because of the evolution of graphics hardware, which is more and more programmable and oriented toward massively parallel rasterization. Such an evolution also leads to a blurring of the boundary between geometry based and image based representation, since more and more geometric information is being encoded in the various texture-based representation to increase rendering fidelity. The techniques used for rendering impostors are strictly related to the issue of height field ray tracing and displacement mapping techniques, a field in which a number of specialized hardware accelerated techniques have been recently presented (e.g., *relief mapping* [5], and *view-dependent displacement mapping* [6]). A very recent evolution of these methods is the BlockMap [7], that compactly represents in a single texture a set of textured vertical prisms with a bounded on-screen footprint that serves as replacement for a set of buildings in city rendering applications. One might argue that the BlockMap representation is more similar to LOD than to impostor approaches, as a BlockMap provides a view-independent, simplified representation of the original textured geometry, provides full support to visibility queries, and, when built into a hierarchy, offers multi-resolution adaptability. Similarly, encoding shape and appearance into a texture is also the goal of geometry images [8], which enables the powerful GPU rasterization architecture to process geometry in addition to images. Finally, there have been a few techniques of applying these image-based rendering techniques to massive models [7], [9], [10].

References

[1] Leonard McMillan and Gary Bishop. Plenoptic Modeling: An Image-Based Rendering System. In *ACM Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 39–46, 1995.

[2] Daniel G. Aliaga and Anselmo A. Lastra. Architectural Walkthroughs Using Portal Textures. In *Proceedings of IEEE Visualization 1997*, pages 355–362, 1997.

[3] François Sillion, George Drettakis, and Benoit Bodelet. Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery. In *Computer Graphics Forum (Proceedings of Eurographics)*, pages 207–218, 1997.

[4] Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. Layered Depth Images. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 231–242, 1998.

[5] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief Texture Mapping. In *ACM Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 359–368, 2000.

[6] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shi-Min Hu, Baining Guo, and Heung-Yeung Shum. View-Dependent Displacement Mapping. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 334–339, 2003.

[7] Paolo Cignoni, Marco Di Benedetto, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, and Roberto Scopigno. Ray-Casted BlockMaps for Large Urban Models Visualization. In *Computer Graphics Forum (Proceedings of Eurographics)*, 2007. To appear.

[8] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry Images. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 335–361, 2002.

[9] Andrew Wilson and Dinesh Manocha. Simplifying Complex Environments Using Incremental Textured Depth Meshes. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRPAH)*, pages 678–688, 2003.

[10] Daniel G. Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kennth E. Hoff III, Tom Hudson, Wolfgang Stürzlinger, Rui Bastos, Mary C. Whitton, Frederick P. Brooks Jr., and Dinesh Manocha. MMR: An Interactive Massive Model Rendering System using Geometric and Image-Based Acceleration. In *SI3D '99: Proceedings of the 1999 Symposium on Interactive 3D Graphics*, pages 199–206, 1999.

Intuitively speaking, we can get fewer cache misses as we store vertices close in the mesh to be also close in the 1D layout. Cache-coherent layouts are layouts constructed in such a way that they minimizes the number of cache misses.

One notable layout method to improve the performance of rasterization is the rendering sequence, which is a sequential list of triangles optimized to reduce the number of GPU vertex cache misses. By computing a good rendering sequence of a mesh, we can expect an up to six times rendering performance improvement. This technique requires the GPU vertex cache size as an input and optimizes the rendering sequence according to this value [20]; therefore, the computed rendering sequence is considered as a *cache-aware* layout.

Recently, *cache-oblivious* mesh layouts have been proposed [18]. These cache-oblivious layouts do not require any specific cache parameters such as block sizes. Instead, these layouts are constructed in such a way that they minimize the *expected* number of cache misses during accessing meshes for block-based caches with various block sizes. Its main advantage compared to other cache-aware layouts is that the user does not need to specify cache parameters such as block size and can get benefits from various levels of memory hierarchies including disk, main memory, and L1/L2 caches. Moreover, the implicit paging system provided by the OS can effectively be coupled with cache-oblivious layouts. Therefore, users can observe the performance improvements without significant modifications on the underlying code and runtime applications. Cache-oblivious layouts have been developed for meshes and bounding volume hierarchies for ray tracing, rasterization, and other geometric applications.

C. Compression Techniques

Mesh compression techniques compute compact representations by reducing redundant data of the original meshes. Mesh compression has been widely researched and good surveys [21] are available. Particularly, triangle strips have been widely used to compactly encode rendering sequences. Triangle strips consist of a list of vertices, which implicitly

encodes a list of triangles. Also, decoding triangle strips for rasterizing triangles can be very efficiently implemented in graphics hardware. However, these representation cannot be directly used for ray tracing, which access underlying meshes in an arbitrary order, unlike the sequential order in rasterization.

There have been efforts to design compressed meshes that also support random accesses for various applications including ray tracing [22], [23]. One of the concepts leading to such techniques is to decompose a mesh into different chunks and compress each chunk independently. Therefore, when a mesh element is required for an application, a chunk containing the required mesh element is decompressed, and the decompressed mesh element can be returned to the application. Recently, the Ray-Strips representation [23] has been proposed for ray tracing. Ray-Strips like triangle strips consist of a list of vertices. Moreover, this vertex list implicitly encodes a list of triangles and a bounding volume hierarchy, which accelerates the performance of ray–triangle intersection tests.

D. Summary

We have discussed three data management techniques, out-of-core techniques, cache-coherent layout methods, and compression techniques, to reduce expensive data access time and, thus, improve the performance of rasterization and ray tracing. Out-of-core techniques mainly focus on reducing disk access time and require specific memory and disk block sizes. Therefore, they can achieve better disk access time compared to cache-oblivious algorithms and layouts. However, out-of-core techniques are usually coupled with an explicit external paging system; therefore, taking advantage of its best performance may require more implementation efforts. On the other hand, cache-oblivious layouts do not require any cache-parameters and work quite well with various GPUs and CPUs having different cache-parameters. Moreover, a user can achieve reasonably high performance without modifying underlying code and applications. Also, by compressing the external data representations and layouts, storage requirements can be drastically reduced and performance of applications is improved.

V. Parallel Processing Techniques

Even when applying the various complexity reduction methods we have visited in the previous sections, rendering of massively complex scenes can still be extremely computationally expensive. Especially in the case of advanced shading, a single CPU/GPU often cannot deliver the performance required for interactive image generation. It becomes thus necessary to combine the computational resources of multiple processing units to achieve a sufficient level of computing power.

Today, parallel computing capabilities are offered at a variety of different hardware and system levels. Examples are SIMD (single instruction multiple data) instructions that can perform vector operations, multiple pipelines in CPUs and GPUs, multi-core architectures, and shared-memory and loosely coupled cluster systems that can contain multiple processors and/or graphics cards.

When focusing on rendering systems using distinct CPUs and/or GPUs, we can distinguish between two main straightforward methods: sort-first and sort-last parallelization strategies. *Sort-first* rendering is based on subdividing screen space into disjoint regions that are rendered independently in parallel by multiple processing units. In a *sort-last* setting, the scene dataset is split into several parts, and is typically distributed amongst separate computing systems individually containing RAM, CPUs, and GPUs. The sub-scenes can be rendered independently, and the results are composed afterwards into the final image. For rasterization this can be accomplished by collecting the content of the individual color and Z-buffers, and choosing the final pixel color based on the nearest depth value for a given pixel. A popular OpenGL oriented system for clusters of workstations that incorporates sort-first, sort-last, and hybrid rendering is Chromium [24]. For a ray tracer a sort-last approach can be handled in a similar way by simply ray tracing the sub-scene images instead of rasterizing them. A different approach for ray tracing is to forward rays from one rendering system to the next if no surface intersection can be found.

A. Data Parallel Rendering

Parallel rendering of a distributed scene database is generally termed *data parallel* rendering (the term sort-last is more related to the composition strategy of the final image). Apart from reducing the complexity of visibility calculations, splitting and distributing a scene between computing sub-systems has another advantage; once the massive scene is decomposed into small chunks, each of which can fit into the available main memory of each sub-system, the overall system can handle highly complex scenes, which would otherwise not fit into the main memory of a single system.

A big disadvantage of such a setup is the difficulty of dealing with advanced shading, as this would require potential access to all parts of the 3D model. In are rasterization based system, data parallel rendering is typically performed using a sort-last image composing mechanism. In case of multi-pass rasterization, various maps (e.g., for shadow calculations) have to be rendered, assembled, and distributed to all hosts. Using ray tracing, mainly sort-first approaches are applied, i.e., an initial primary ray is sent to the sub-system that hosts the region of the scene the ray enters first. Rays thereafter have to be propagated between the individual sub-systems, which usually results in a high communication overhead, especially for secondary rays. In addition, using pure data parallel scheduling of rendering tasks typically does not allow for handling load-imbalances caused by changing viewpoints.

B. Demand Driven Rendering

When pursuing a screen space subdivision (sort-first) approach, a straightforward way is to use a static partitioning scheme, where the image space is broken into as many fixed-size regions as there are rendering client machines. Another alternative is to split the image into many small, often quadrangular regions called *tiles*, which are assigned to the available rendering clients (i.e., processing units). Depending on the part

of a scene that is visible in a tile, computation time for each tile can vary strongly across the image plane. This can lead to bad client utilization, and thus result in a poor scalability if tiles would be statically assigned to the rendering clients. Therefore, a better alternative is to employ a *demand driven* scheduling strategy, by letting the clients themselves ask for work. As soon as a client has finished a tile, it sends its results back to the master process (which is responsible for assembling and displaying the image), and request the next unassigned tile. This leads to an efficient load balancing, and for most scenes to an almost linear scalability in the number of rendering clients.

C. Distributed Rendering

In contrast to shared-memory environments, where a number of CPUs can simultaneously access a virtually single contiguous chunk of main memory, distributed systems contain a number of physically separated computing systems communicating over an interconnection network. In such an environment, typically one dedicated machine serves as host for a master process. The master is responsible for distributing the rendering workload among the remaining client machines, and for assembling and displaying the generated image.

Particularly in situations where clients render massive datasets in an out-of-core manner, it is important to consider spatio-temporal coherence. In order to make best use of caches on client machines, image tiles should be assigned to the same clients in subsequent frames whenever possible.

To hide latencies, rendering, network transfer, image display, and updating scene settings should be performed asynchronously. For example, while image data for frame N is transferred, the clients already render frame $N + 1$, whereas the application can specify and send updated scene settings for frame $N + 2$.

D. Summary

Modern CPUs and GPUs increasingly feature parallel computing capabilities. One of the most important computational trends is the growing numbers of CPU cores and GPU processing units, as physical limitations in stepping up clock rates and reducing sizes of integrated circuit structures become more and more eminent. Therefore, future hardware will make it possible to render today's massive models on standard computing systems, but scene complexity is also expected to keep rising for quite some time to come. For such extremely complex scenes, it is required to combine the computational power of multiple computing systems to enable interactive rendering and sophisticated shading.

In this section we only dealt with parallel rendering. However, parallel installations described in this section can be equally applied to speed up precomputation tasks, like, e.g., building spatial index structures or computing level-of-detail representations.

VI. System Issues

Rendering high-quality representations of complex models at interactive rates requires not only to carefully craft algorithms and data structures, but also to combine the different components in an efficient way. This means that the different solutions illustrated in the previous sections must be carefully mixed and matched in a single coherent system able to balance the competing requirements of realism and frame rates. No single standard approach presently exists, and the different solutions developed to date all have their advantages and drawbacks. In the following, we briefly illustrate how a few representative state-of-the-art systems work.

A. Visibility Driven Rasterization

One of the important uses for massive model rendering is the exploration of models with a high depth complexity. These application test cases include architectural walkthroughs and explorations of large CAD assemblies. In these situations, occlusion culling is often the most effective visible data reduction techniques. A number of systems have thus been developed around efficient visibility queries.

A system that efficiently makes use of the occlusion query capabilities of modern graphics hardware is the **Visibility Guided Rendering** [17] (VGR) system. VGR organizes the scene in a hierarchy of axis-aligned bounding boxes. For each internal node of the resulting tree a splitting plane along one of the three primary axes (like in a regular kd-tree) is stored, which is used to traverse the scene in a front-to-back order. In a preprocessing step the tree is generated top-down, while trying to keep the edges of boxes of equal size, to optimize culling efficiency for different viewing angles. Recursively subdividing the scene is terminated once a node contains about 2000 to 4000 triangles.

The faces of boxes associated with nodes are directly used as test geometry for hardware occlusion queries. Ideally, traversal of the bounding box hierarchy would be performed depth-first, selecting children of a node in a front-to-back order. However, with occlusion queries being done on the GPU and tree traversal on the CPU, this would result in a stop-and-go behavior. As explained in section III-C, the solution is to carefully schedule queries by exploiting spatial and temporal coherence. Thus, to allow for running occlusion queries in parallel to tree traversal, VGR maintains a queue of query requests, which can be asynchronously processed by the GPU. Rather than carrying out visibility checks in a depth-first order, the queue is filled based on a more breath-first traversal order (see Figure 11a). This results in a slightly reduced culling efficiency, since farther nodes might be wrongly classified as visible as not all nearer nodes have been rendered. However, it avoids introducing GPU stalls, thus increasing overall performance.

VGR also maintains a list of leaf nodes that were visible in the previously rendered frame. In successive frames new visible nodes are added to the list while others become invisible. The nodes of this list are rendered first in the current frame. The intention is to fill the Z-buffer before the first occlusion query takes place, thus exploiting frame-to-frame coherence. Visibility information for leaf nodes from the previous frame is propagated up the tree, which makes it then possible to exclude sub-trees from traversal and visibility testing (Figure 11b). Not every leaf is tested for visibility in

each frame. Occlusion testing is performed every n frames for the leaf nodes contained in the list.

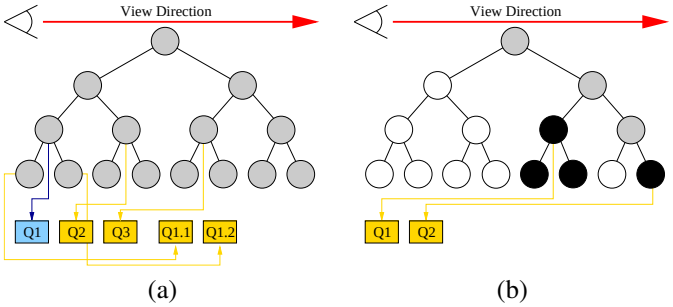


Fig. 11. **Visibility Guided Rendering (VGR).** (a) Scheduling VGR occlusion queries. A query queue is maintained, and is filled with requests based on a breadth-first traversal, in order to avoid GPU lag times. (b) Exploiting spatio-temporal coherence. Leaf nodes visible in the previous frame are marked white and are rendered first, invisible nodes are marked black. This information is propagated up the tree, only black marked sub-trees need to be tested for visibility.

For handling complex scenes, VGR can render in an out-of-core mode. To this end, it maintains a two-level caching architecture, where the graphics card memory (VRAM) serves as first-level cache, and the RAM of the host machine as second-level cache. Memory is managed using a least recently used (LRU) cache eviction strategy. The VRAM is subdivided into a small number of large slices containing OpenGL vertex and index buffers. Slices are filled from the front to the back with data from visible leaf nodes. Once no more free slices are left, the slice least recently used is completely emptied and refilled.

The VGR system also incorporates a simple LOD mechanism. Before rendering a visible node, its screen space projection area is determined. If the area is smaller than a pixel, the system resorts to point rendering for this node. In addition, it is further possible to randomly thin out the point cloud for very distant nodes.

While the VGR system employs online visibility queries, other systems make use of visibility information computed in a preprocessing phases and stored with the model. A representative system of this category is **iWalk**, which is constructed around a multi-threaded out-of-core rasterization method.

iWalk can support high-resolution (4096×3072) and multi-tiled displays by employing sort-first parallel out-of-core rendering. iWalk decomposes an input model with an octree. For construction, since an input model typically does not fit into the available memory, iWalk breaks the model into small sections, each of which can fit into main memory. Then, iWalk incrementally constructs an octree by processing each section in a separate pass, and merges the final result into a single octree.

iWalk can be integrated with approximate or conservative visibility culling and employs speculate prefetching considering visibility events, which are very hard to deal with. To do that, a visibility coefficient for each octree node is computed. A visibility coefficient measures how much geometry in an octree node is likely to be occluded by other geometry. At runtime, iWalk predicts visibility events based on visibility coefficients stored in the octree nodes. This feature allows the system to pre-fetch the geometry which is likely to be accessed in a next frame, and thus reduces expensive loading time of newly visible geometry.

iWalk also uses multi-threading to concurrently perform visibility prefetching, rendering, and out-of-core management. Since disk operations are very expensive and have high latency, multi-threading of different tasks achieves higher CPU utilization and thus improves the overall performance of rendering massive models.

B. Real-Time Ray Tracing

Another class of systems heavily relying on efficient visibility culling is that of systems built around real-time ray tracing kernels.

A very advanced system of this kind is the **OpenRT** real-time ray tracer [2], [25], a renderer originally conceived to deliver interactive ray tracing performance on low-cost clusters of commodity PCs. It can, however, also be run in a shared-memory environment. OpenRT uses a two-level kd-tree hierarchy as spatial index. A scene can consist of several independent objects composed of geometric primitives, where each object has its own local kd-tree. The bounding boxes of such objects are organized in a top-level kd-tree. On the one hand, this allows for a limited form or rigid-body motion, since only the top-level tree needs to be rebuilt once objects move. On the other hand, it enables efficient instancing of objects as the top-level tree can contain multiple references and corresponding transformation matrices to a single object. Kd-tree construction makes use of cost prediction functions to estimate optimal splitting plane positions.

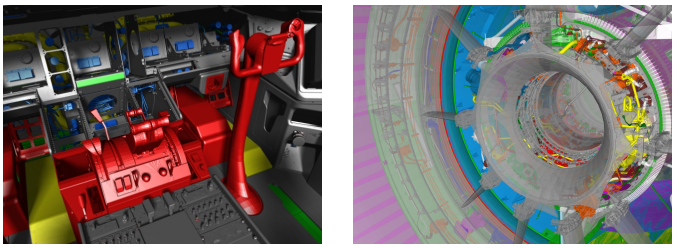
As long as the scene description can fit completely into main memory, even visually highly complex scenes can be handled due to the logarithmic time complexity of ray tracing. An example can be seen in Figure 1d. Using tile-based demand driven rendering on multiple CPUs, the depicted landscape scene can be interactively explored without having to incorporate explicit occlusion culling or level-of-detail methods.

OpenRT incorporates a custom memory management subsystem that can deal with scenes larger than physical memory in an out-of-core fashion. The whole dataset, including acceleration structures, etc. is mapped from disk into virtual address space using the operating system's memory mapping facilities (e.g., Linux `mmap()`). Making use of the OS memory mapping system provides an automatic demand paging scheme, taking care that data is loaded into main memory as soon as it is accessed. However, in order to avoid stalling due to page faults during ray traversal and intersection, it is necessary to detect whether or not memory referenced by a pointer is actually in core memory. To this end, a hash-table is maintained that records which pages of the model have already been loaded. In case of a potential page fault, tracing a ray is canceled, while the missing memory page is scheduled to be loaded by an asynchronously running fetcher thread.

In case of a canceled rays several strategies can be applied. For smaller models (in the range of a few dozen million triangles), where missing data can be loaded during a single

frame, rays can be suspended and later resumed once the data becomes available in memory. For larger models, simplified representations that can fully fit into memory are used as a substitute. It should be noted that this is only necessary to bridge loading time, but not to reduce the visual complexity. Simplified data is only used while fully-detailed data is being loaded.

OpenRT can use different types of surface shaders in a plug-and-play manner, which makes it possible to include different types of shading and lighting effects, e.g. soft shadows or transparency (see Figure 12) that can help to better visualize the model structure.



(a) (b)

Fig. 12. **Examples of surface shading effects.** (a) Soft shadow effects in a Boeing 777 cockpit providing a better impression of the relative placement of components. (b) Adding transparency can help in understanding the complex part arrangement in the engine nacelle.

In contrast to OpenRT, which was originally conceived for distributed PC cluster rendering, the **Manta** Open Source Interactive Ray Tracer [26] has be designed from scratch for shared-memory multi-processor systems.

Manta's architecture consists of two fundamental components, a parallel pipeline executed by all threads synchronously, and a set of modular components organized in a rendering stack, which is executed by each thread asynchronously. The pipeline is organized by dividing rendering tasks based on their parallel load characteristics into inherently balanced, imbalanced, and dynamically balanced categories. All rendering threads perform each task on each frame in the pipeline during a stage. Dynamically load balanced tasks are executed last so any imbalance introduced earlier can be smoothed out and processor stalls between stages are avoided. A basic Manta rendering pipeline consists of ray tracing and image display. Since image display is usually only performed by one thread, this task is imbalanced. In the Manta control loop, the previous frame is displayed first and then the rendering stack is invoked to perform ray tracing of the current frame. After the responsible thread completes display, it joins the rendering threads.

Thread safe state changes in Manta are performed using callbacks to simple application defined functions called transactions. These transactions are stored in a queue and processed between pipeline stages when all threads are synchronized. Ideally, each transaction performs a very small state change, given the high performance (relative to rendering rates) of modern barriers, transaction performance is higher than individually locking shared state. Additionally, Manta supports a variety of scheduling techniques that allow callbacks to be invoked at specific times or frame numbers.

The transaction facility allows Manta to be easily embedded in other applications. Thread safe state changes are performed by defining callback functions and then passing the functions to Manta using transactions. Changes can be entirely application specific from simple camera movements to scene graph or material property changes.

C. LOD Based Mesh Rasterization

Relying on efficient visibility determination alone is not sufficient to ensure interactive performance for highly complex scenes with a lot of very fine details, since, in order to bound the amount of data required for a given frame, a prefiltered representation of details must also be available. When dealing with very large detailed meshes, such as those generated by laser scanners, some of the highest performance systems to date are based on the rasterization of multi-resolution point- or vertex-hierarchies constructed off-line through a geometric simplification process.

For instance, the **Quick-VDR** [13] system is constructed around a dynamic LOD representation, and achieves interactive performance by combining various techniques mentioned in earlier sections. To efficiently provide view-dependent rendering of massive models based on dynamic LODs, Quick-VDR proposed a clustered hierarchy of progressive meshes (CHPM). The CHPM consists of two parts: a cluster hierarchy and progressive meshes. Quick-VDR represent the entire dataset as a hierarchy of clusters, which are spatially localized mesh regions. Each cluster consists of a few thousand triangles. The clusters provide the capability to perform coarse-grained view-dependent (or selective) refinement of the model. They are also used for visibility computations and out-of-core rendering. Then, Quick-VDR precomputes a simplification of each cluster and represents a linear sequence of edge collapses as a progressive mesh (PM). The PMs are used for fine-grained local refinement and to compute an error-bounded simplification of each cluster at runtime. Also, explicit dependencies between clusters are maintained in order to guarantee crack-free simplifications on the mesh. The major benefit of the CHPM representation is its ability to provide efficient, but effective dynamic LODs for massive models by combining coarse-grained refinement based on clusters and fine-grained local refinement providing smooth transitions between different LODs.

Quick-VDR can render massive models without a significant loss of image quality, although all the data structures cannot fit into the available main memory. Also, conservative visibility culling implemented with hardware-accelerated occlusion queries are integrated with rendering with the CHPM representation. A major downside of this method is a relatively low GPU vertex cache utilization during rendering dynamic LODs compared to rendering static LODs. However, this low cache utilization was addressed by employing cache-oblivious mesh layouts for ordering of triangles and vertices of dynamic LODs.

The **Adaptive TetraPuzzles** [12] (ATP) system also introduced a solution based on a patch-based multi-resolution data structure, from which view-dependent conforming mesh

representations can be efficiently extracted by combining precomputed patches. In the ATP case, however, the system does not need to maintain explicit dependencies, since the method uses a conformal hierarchy of tetrahedra generated by recursive longest edge bisection to spatially partition the input mesh. In this case, each tetrahedral cell contains a precomputed simplified version of the original model, which is constructed off-line during a fine-to-coarse parallel out-of-core simplification of the surface contained in diamonds (sets of tetrahedral cells sharing their longest edge). Appropriate boundary constraints are introduced in the simplification process to ensure that all conforming selective subdivisions of the tetrahedron hierarchy lead to correctly matching surface patches. At run-time, selective refinement queries based on projected error estimation are performed on the external memory tetrahedron hierarchy to rapidly produce view-dependent continuous mesh representations by combining precomputed patches.

Using coarse grained LODs also serves in the Quick-VDR and ATP systems for out-of-core management, which is done by explicitly maintaining LRU caches of mesh patches.

D. Switching to Alternate Rendering Primitives

Adaptive meshing systems such as those discussed above tend to perform best for highly tessellated surfaces that are otherwise relatively smooth and topologically simple, since it becomes difficult, in other cases, to derive good "average" merged properties. Since performing iterative mesh simplification does not provide visually adequate simplifications when dealing with complicated topology, geometry and appearance, systems have started to appear that use alternate rendering primitives for data prefiltering.

The **Far Voxels** [27] system, for instance, exploits the programmability and batched rendering performance of current GPUs, and is based on the idea of moving the grain of the multi-resolution surface model up from points or triangles to small volumetric clusters, which represent spatially localized dataset regions using groups of (procedural) graphics primitives. The clusters provide the capability of performing coarse-grained view-dependent refinement of the model and are also used for on-line visibility culling and out-of-core rendering.

Figure 13 provides an overview of the approach. To generate the clusters, the model is hierarchically partitioned with an axis-aligned BSP tree. Leaf nodes partition full resolution data into fixed triangle count chunks, while inner nodes are discretized into a fixed number of cubical voxels arranged in a regular grid.

Finding a suitable voxel representation is challenging, since a voxel region can contain arbitrarily complex geometry. To simplify the problem, the method assumes that each inner node is always viewed from the outside, and at a distance sufficient to project each voxel to a very small screen area (say, below one image pixel). This constraint can be met with a suitable view-dependent refinement method, that refines the structure until a leaf is encountered or the image of each voxel is small enough. Under this condition, a voxel always subtends a very small viewing angle, and a purely direction dependent representation of shading information is thus sufficient to produce

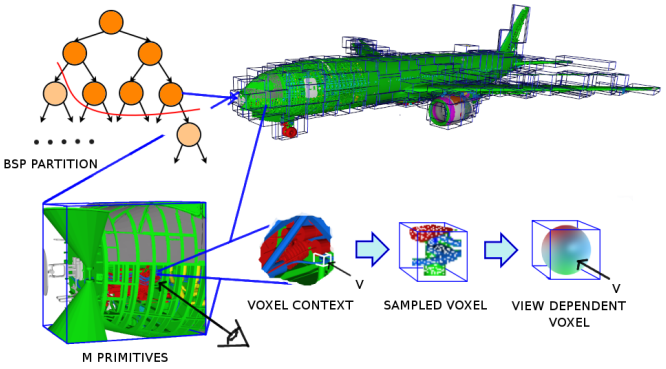


Fig. 13. **Far voxels overview.** The model is hierarchically partitioned with an axis-aligned BSP tree. Leaf nodes are rendered using the original triangles, while inner nodes are approximated using view-dependent voxels.

accurate visual approximations of its projection. To construct a view-dependent voxel representation, the method employs a visibility aware sampling and reconstruction technique. First, a set of shading information samples is acquired by ray casting the original model from a large number of appropriately chosen viewing positions. Each sample associates a reflectance and a normal to a particular voxel observation direction. Then, these samples are compressed to an analytical form that can be compactly encoded and rapidly evaluated at run-time on the GPU to compute voxel shading given a view direction and light parameters. At rendering time, the volumetric structure, maintained off-core, is refined and rendered in front-to-back order, exploiting vertex shaders for GPU evaluation of view-dependent voxel representations rendered as point primitives, hardware occlusion queries for culling occluded sub-trees, and asynchronous I/O for avoiding out-of-core data access latencies. Since the granularity of the multi-resolution structure is coarse, data management, traversal and visibility culling costs are amortized over many graphics primitives, and disk/CPU/GPU communication can be optimized to fully exploit the complex memory hierarchy of modern graphics PCs.

The resulting technique has proven to be fully adaptive and applicable to a wide range of model classes, that include very detailed colored objects composed of many loosely connected interweaving detailed parts of complex topological structure (see Figure 14). Its major drawbacks are the large preprocessing costs and the aliasing and transparency handling problems due to the point splatting approach.

Even if it is often neglected, finding good LOD representations is also of primary importance for ray tracing systems. Even if visibility determination in ray tracing has logarithmic growth-rate, due to the use of acceleration hierarchies, the runtime access pattern of massive model ray tracing can be very incoherent. For instance, most portions of the hierarchies and meshes can be accessed and traversed during ray–triangle intersection tests when generating zoomed out views of very large models. Therefore, in the absence of a suitable LOD representation, the working set size of ray tracing can be very high, and when it becomes bigger than the available main memory, the performance of ray tracing is significantly

Fig. 14. **Far Voxels Rendering Example.** A 1.2 billion triangles scene interactively inspected on a large scale stereoscopic display driven by single PC, which renders two 1024×768 images per frame with a 1 pixel tolerance.

degraded.

To address this issue, the **R-LOD** [28] system has introduced a LOD representation for ray tracing tightly integrated with kd-trees. Specifically, a R-LOD consists of a plane with material attributes (e.g., color), which is a drastic simplification of the descendant triangles contained in an inner node of the kd-tree, as shown in Figure. 15, and is similar to one of the shaders employed by the Far Voxels system. Each R-LOD is also associated with a surface deviation error, which is used to quantify the projected screen space error at runtime.

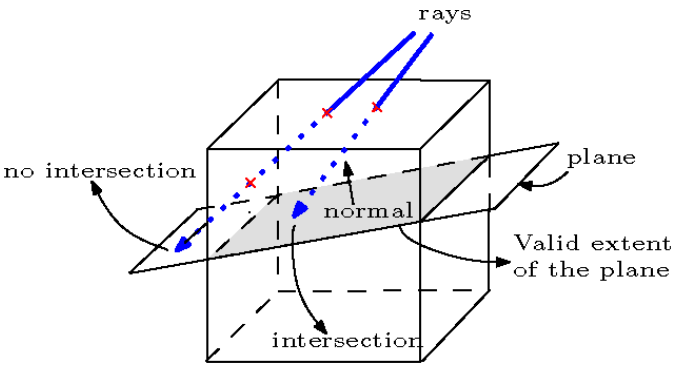


Fig. 15. **R-LOD Representation.** A R-LOD consists of a plane with material attributes. It serves as a drastic simplification of triangle primitives contained in the bounding box of the sub-tree of a kd-tree node. Its extent is implicitly given by its containing kd-node. The plane representation makes the intersection between a ray and a R-LOD very efficient and results in a compact representation.

If a R-LOD representation of a kd-node has enough resolution for a ray according to a LOD metric, further hierarchy traversal for ray-triangle intersection tests stops and performs ray-LOD intersection tests. As a major benefit of this method, it can reduce the working set size by traversing smaller amount of hierarchy and providing LOD representations for the input model. As a result, it can drastically improve the performance of ray tracing massive models. Moreover, the R-LOD representation can improve the SIMD utilization of ray-coherence techniques. This is mainly possible because an LOD representation is likely to be chosen for a kd-node if hierarchy traversals of rays in a ray packet are getting to show low-coherence.

As a downside of this approach, this method does not provide the full LOD solutions for arbitrary rays, especially for non-linear transformations for refractions and reflections off of curved surfaces. Moreover, in some cases, viewers can observe visual artifacts, which is a very serious problem for ray tracing.

E. Summary

Today, a broad variety of massive model rendering systems exists that allow for the interactive display of very large models. While this rapid survey is by no means exhaustive, the analyzed systems provide a good sampling of today's available options to implementing a state-of-the-art rendering system. From this overview it can be seen that rendering systems are typically non-trivial frameworks that need to incorporate many techniques, usually from all of the in the previous sections presented categories. Today, no universal system exist that can handle all massive models application scenarios. As it should be clear from this brief analysis many options exists, but at the same time, many more similarities among systems exist than it can appear at a first look, since in the end, all systems pick from a the same bag of techniques.

VII. Conclusion

In this article, we have examined various techniques of improving rendering performance for massive 3D synthetic environments. Such massive models are increasingly common as a result of the phenomenon known as information explosion in more general contexts. This trend is doomed to continue: for instances, just consider that today's massive model scenes have a really small complexity compared to real live environments.

It can be argued that, while a number of applicable solutions exists, efficient processing of large scale datasets for interactive visualization is a challenging open-ended problem, that must be continuously addressed. By learning from the last decade of research in this field, and taking into account the current hardware evolution, it is possible to find some common guidelines, and draw some conclusions on how to realize current systems and plans for the future.

We have seen that, at the broad level, current massive model rendering approaches can be classified into rasterization or ray tracing methods. We argue that, when it comes to dealing with massive datasets, the underlying issues are somewhat similar. All the methods have to deal with the same data management and filtering problems and are converging towards proposing similar solutions, based on spatial indexing, data reduction techniques, and data management methods. Even though in the past the ray tracing and rasterization fields have independently developed their own approaches, it is likely that future systems will incorporate hybrid approaches, in particular as graphics hardware is becoming more and programmable and will allow for executing rasterization and ray tracing side-by-side.

One point that is now clear in massive model visualization is that, while large scale rendering problem cannot just be

solved by waiting for more powerful hardware, hardware trends dictate which methods are successful and which are doomed to be practically inefficient. The challenge is thus in designing methods able to capture as much as the performance growth as possible. Current multi-core CPU systems and GPUs excel at massively parallel tasks with good memory locality, since the gap between computation performance and bandwidth throughout the memory hierarchy is growing. For this reason, we expect that methods for carefully managing working set size, ensuring coherent access patterns, as well as data parallel techniques will increasingly gain importance.

Up until very recently, the various problems handled by a renderer were independently solved. It is now increasingly clear that there are important couplings between the different components of a massive model renderer. For instance, generating good levels of details for very complex models requires visibility information. At the same time, the availability of a multi-resolution model typically increases data size, but is essential to increase memory coherence and reduce working set size. At present time, good solutions are available for a restricted number of situations and restricted classes of models.

Although there has been a lot of advances on massive model rendering techniques for static models, there have been relatively less research efforts on dealing with time-varying, dynamic, or animated models. Since these dynamic models are getting easier to model and capture, it is expected that there will be higher demand for efficient dynamic model management techniques. Currently, there is a new trend in investigating how to rapidly build and update acceleration structures, and how to best trade culling efficiency against construction time. While this research is so far mainly focused on much smaller dynamic models (see, e.g., [29]), the results are equally important when dealing with massively complex scenes, where such methods are not only applicable for animation, but also for fast preprocessing. These incremental methods need also be extended to tasks other than spatial indexing, e.g., generating LODs.

Finally, very little research has been done on how to adapt advanced shading and light transport simulation techniques to massively complex scenes, especially in a real-time setting. Although a tremendous amount of research targeting photo-realistic image synthesis has been carried out in the last decades, such techniques cannot easily be applied to massive environments. While handling pure visibility in arbitrarily sized models still remains challenging, it is likely that additional efforts will be made to also include more sophisticated illumination effects.

ACKNOWLEDGMENTS

The authors would like to thank Fabio Marton and Abe Stephens for their help. Source 3D datasets are provided by and used with permission of the Boeing Company, the Digital Michelangelo Project, the Lawrence Livermore National Laboratory, the Georgia Institute of Technology, and the University of Konstanz. The Sponza Atrium scene has been modeled by Marko Dabrovic. This work was partially supported by the Italian Ministry of University and Research under grant CYBERSAR, and by a KAIST seed grant.

References

[1] Philip Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination, Second Edition*. A K Peters, 2006.

[2] Ingo Wald, Timothy J. Purcell, Joerg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics 2003 State of the Art Reports*, 2003.

[3] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 434–444, 2005.

[4] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. In *ACM Transaction of Graphics (Proceedings of ACM SIGGRAPH)*, pages 1176–1185, 2005.

[5] David P. Luebke. A Developer's Survey of Polygonal Simplification Algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, 2001.

[6] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of Triangle Meshes. In *ACM Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 65–70, 1992.

[7] Hugues Hoppe. Progressive Meshes. In *ACM Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 99–108, 1996.

[8] Michael Garland and Paul S. Heckbert. Surface Simplification Using Quadric Error Metrics. In *ACM Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 209–216, 1997.

[9] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink. Large Mesh Simplification using Processing Sequences. In *Proceedings of IEEE Visualization 2003*, pages 465–472, 2003.

[10] Carl Erikson, Dinesh Manocha, and William V. Baxter III. HLODs for Faster Display of Large Static and Dynamic Environments. In *SI3D '01: Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 111–120, 2001.

[11] Leila De Floriani, Paola Magillo, and Enrico Puppo. Efficient Implementation of Multi-Triangulations. In *Proceedings of IEEE Visualization 1998*, pages 43–50, 1998.

[12] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Adaptive TetraPuzzles: Efficient Out-of-Core Construction and Visualization of Gigantic Multiresolution Polygonal Models. In *ACM Transactions on Graphics (Procceedings of ACM SIGGRAPH)*, pages 796–803, 2004.

[13] Sung-Eui Yoon, Brian Salomon, Russel Gayle, and Dinesh Manocha. Quick-VDR: Interactive View-Dependent Rendering of Massive Models. In *Proceedings of IEEE Visualization 2004*, pages 131–138, 2004.

[14] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[15] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering, Second Edition*. A K Peters, 2002.

[16] Jirí Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, 2004.

[17] Mathias Heyer, Sebastian Pfützer, and Beat Brüderlin. Visualization Server for Very Large Virual Reality Scenes. In *4. Paderborner Workshop Augmented & Virtual Reality in der Produktentstehung*, 2005.

[18] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-Oblivious Mesh Layouts. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 886–893, 2005.

[19] Cláudio T. Silva, Yi-Jen Chiang, Jihad El-Sana, and Peter Lindstrom. Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics. In *Course Notes IEEE Visualization 2002 Tutorial #4*, 2002.

[20] Hugues Hoppe. Optimization of Mesh Locality for Transparent Vertex Caching. In *ACM Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 269–276, 1999.

[21] Pierre Alliez and Craig Gotsman. Recent Advances in Compression of 3D Meshes. In N.A. Dodgson, M.S. Floater, and M.A. Sabin, editors, *Advances in Multiresolution for Geometric Modelling*, pages 3–26. Springer, 2005.

[22] Sung-Eui Yoon and Peter Lindstrom. Random-Accessible Compressed Triangle Meshes. In *Proceedings of IEEE Visualization 2007*, 2007. To appear.

[23] Christian Lauterbach, Sung-Eui Yoon, and Dinesh Manocha. RayStrips: A Compact Mesh Representation for Interactive Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing*, 2007. To appear.

[24] Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, pages 693–702, 2002.

[25] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering)*, pages 81–92, 2004.

[26] Abraham Stephens, Solomon Boulos, James Bigler, Ingo Wald, and Steven Parker. An Application of Scalable Massive Model Interaction using Shared-Memory Systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, pages 19–26, 2006.

[27] Enrico Gobbetti and Fabio Marton. Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 878–885, 2005.

[28] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-LODs: Fast LOD-Based Ray Tracing of Massive Models. *The Visual Computer*, 22(9–11):772–784, 2006.

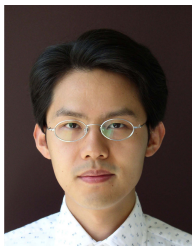
[29] Ingo Wald, William R Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, 2007. To appear.

Andreas Dietrich is a research assistant and Ph.D. student at the Computer Graphics Group at Saarland University. His research focuses on real-time ray tracing for VR applications, natural phenomena rendering, and on interactive massive model visualization. He received a masters degree (Dipl.-Technoinform.) in computer science from Kaiserslautern University in 2001.

Address: Andreas Dietrich, Computer Graphics Group, Saarland University, Campus E1 1, 66123 Saarbrücken, Germany, dietrich@cs.uni-sb.de

Enrico Gobbetti is the director of the Visual Computing group at the CRS4 research center, Italy. He holds Engineering (1989) and Ph.D. (1993) degrees in computer science from the Swiss Federal Institute of Technology in Lausanne. His research spans many areas of computer graphics and is widely published in major journals and conferences. Many of the technologies developed by his group have been used in as diverse real-world applications as internet geoviewing, scientific data analysis, and surgical training.

Address: Enrico Gobbetti, Visual Computing Group, CRS4 - Center for Research, Development, and Advanced Studies in Sardinia, Sardegna Ricerche Edificio 1, Loc. Pixina Manna, C. P. 25, I-09010 Pula (CA), Italy, gobbetti@crs4.it



Sung-Eui Yoon is currently an assistant professor at Korea Advanced Institute of Science and Technology (KAIST). He received the B.S. and M.S. degrees in computer science from Seoul National University in 1999 and 2001 respectively. He received his Ph.D. degree in computer science from the University of North Carolina at Chapel Hill in 2005. He was a postdoctoral scholar at Lawrence Livermore National Laboratory. His research interests include visualization, interactive rendering, geometric problems, and cache-coherent algorithms and layouts.

Address: Sung-Eui Yoon, Department of Computer Science, KAIST - Korea Advanced Institute of Science and Technology, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, South Korea, sungeui@cs.kaist.ac.kr