vLOD: High-Fidelity Walkthrough of Large Virtual Environments

Jatin Chhugani, Budirijanto Purnomo, Shankar Krishnan, Jonathan Cohen, Suresh Venkatasubramanian, David Johnson, and Subodh Kumar, *Member*, *IEEE Computer Society*

Abstract—We present visibility computation and data organization algorithms that enable high-fidelity walkthroughs of large 3D geometric data sets. A novel feature of our walkthrough system is that it performs work proportional only to the *required* detail in *visible* geometry at the rendering time. To accomplish this, we use a precomputation phase that efficiently generates per cell *vLOD*: the geometry visible from a view-region at the right level of detail. We encode changes between neighboring cells' vLODs, which are not required to be memory resident. At the rendering time, we incrementally construct the vLOD for the current view-cell and render it. We have a small CPU and memory requirement for rendering and are able to display models with tens of millions of polygons at interactive frame rates with less than one pixel screen-space deviation and accurate visibility.

Index Terms—Interactive walkthrough, levels of detail, visibility computation, compression.

1 Introduction

Computer modeling is essential to visualize data in fields such as engineering, flight training, military exercises, medical and other simulations. Geometric models are used for training, planning, design, and other analyses in these fields. It is not surprising then that models have been quickly growing in complexity and richness, making interactive walkthrough of these models challenging. Fortunately, the 3D model-display capability of widely available graphics hardware has been growing steadily. However, the sizes of models have grown faster. Models like ships, factories, and cities may require tens of millions of polygons or more and may not even fit in the main memory. In practice, graphics hardware can barely display such models once in a few seconds, but interactive walkthroughs require the display of over 10–20 frames every second.

As a result, CPU-based algorithms are commonly used to reduce the number of polygons sent to the hardware pipeline. This typically includes some preprocessing, such as model simplification, as well as some per frame computation. Often this CPU load is still too high for interactive display. Furthermore, in applications where simulations such as collision detection need to be performed in conjunction with the visualization, this CPU overload can become especially acute.

Main Contributions. Our main contribution is a technique that enables *high-fidelity* walkthrough of *large* models using *low CPU load*. These three desirable properties have been tough to achieve simultaneously in any system so far.

- J. Chhugani, B. Purnomo, J. Cohen, and S. Kumar are with Johns Hopkins University, 224 NEB, 3400 Johns Hopkins University, Baltimore, MD 21218. E-mail: {jatinch, bpurnomo, cohen, subodh}@cs.jhu.edu.
- S. Krishnan, S. Venkatasubramanian, and D. Johnson are with AT&T Labs, 180 Park Ave., Florham Park, NJ 07932. E-mail: {krishnas, suresh, dsj}@research.att.com.

Manuscript received 26 Aug 2003; revised 15 May 2004; accepted 16 June 2004. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0076-0803.

In our approach, these strengths are combined by precomputing levels of detail and visibility in an integrated way. Done naively, this can require prohibitively large computation time and storage. We address these issues using the following algorithms:

- **From-region occlusion computation.** Our algorithm is simple to implement, yet very effective. It is scalable and accelerates well on graphics hardware. It admits concave occluders and does not require heuristic preselection of a small number of occluders.
- **Object Reordering.** We provide a formulation of the disk layout problem for out-of-core model geometry and its solution. This customized disk layout helps reduce the time taken to load data needed in any given frame.
- **Visibility mask compression.** Raw encoding of visibility information for an entire viewing space is prohibitively large. We provide a new customized compression algorithm to achieve 34x compression on average, significantly better than prior approaches. In addition, the decompression speed is extremely fast.

Using a combination of these algorithms, we obtain *interactive display* of large *out-of-core* models. Furthermore, we are able to guarantee *high-fidelity* by employing LOD with conservative error bounds at an extremely low setting of only one pixel screen-space deviation and by using a *fully conservative* 3D visibility algorithm. We incur a *low CPU load* by precomputing the *composition* of LOD and visibility to generate *vLOD*, a representation of the visible geometry at the correct level of detail for all locations in the desired viewing space. This eliminates the need for expensive visibility computations at run time, reducing the online algorithm to a basic caching system for retrieving the appropriate visibility information and associated geometry. The precomputation parallelizes very well and can be performed efficiently using a cluster of workstations.

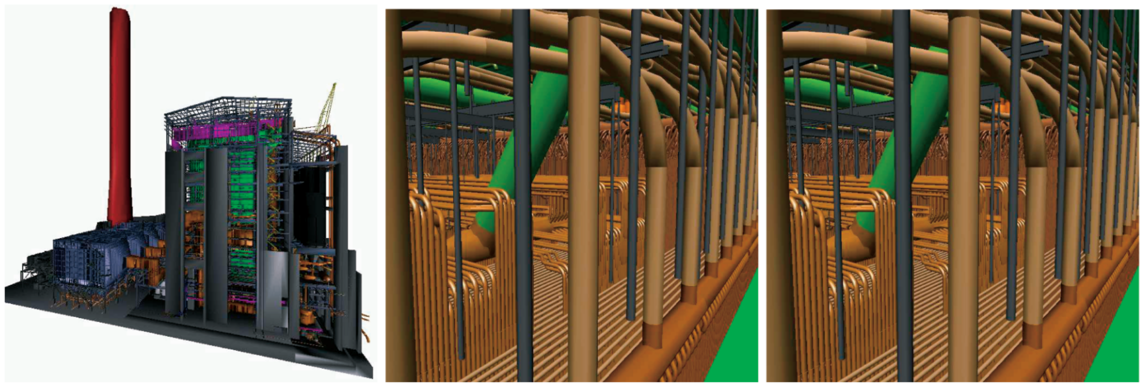


Fig. 1. (a) UNC power plant model with 13 million triangles. (b) Furnace area of the original model with view frustum culling alone (6.2 million triangles). (c) Furnace area with vLOD, conservative from-region visibility, and less than one pixel screen space deviation due to LOD (3.2 million triangles). Notice that rendering with vLOD preserves the fidelity of the scene.

2 Previous Work

Much of the related research has concentrated on visibility precomputation and detail reduction.

Visibility computation is a challenging problem and has been the focus of significant research. It usually includes occlusion culling, back-face culling, and view-frustum culling. We perform all these in our system. Object-based visibility computation algorithms hierarchically compute the visibility for a group of triangles from the given viewpoint [9], [20], [22], [26], [47]. On the other hand, from-region (or cell-based) visibility algorithms [7], [13], [36], [40], [44], [45] precompute conservative visibility from a region of space. At rendering time, objects visible from the view cell may be further refined for each viewpoint or simply sent to the graphics pipeline to generate the final image. From-region visibility computation has become popular recently due to its relatively low cost at the rendering time.

An exact and effective from-region visibility algorithm could be based on the computation of aspect graphs [17] or visibility complex [11], [12]. However, with size complexity of $O(n^9)$ and $O(n^4)$, respectively, and similarly high computational complexity, such approaches are infeasible for large models. The more practical approach is to subdivide space, say using an octree, and precompute visibility from each cell in the space. Exact visibility from a given cell requires ray intersection tests in a four-dimensional space [33] and is too slow. Hence, only conservative overestimation of the visible set from each given view-cell is practical. These methods retain *some* hidden geometry but do not cull *any* visible geometry. Standard graphics Z-buffer hardware eventually displays only the visible geometry, discarding *every* hidden geometry.

Of the recent conservative 3D occlusion computation algorithms, only the ray-space factorization algorithm [28] comes close to speeds that are necessary for precomputing visibility of models with tens of millions of polygons from a large number of view-regions. It is designed to work well on "3D-ϵ" scenes, where the vertical depth complexity is significantly lower than the horizontal depth complexity. It computes the visibility of each cell in several seconds on average. In comparison, our technique is simpler and faster and can compute the visibility for each cell in less than a second on average.

Geometry simplification methods precompute a hierarchy of levels of detail (LODs) and, at rendering time, traverse a data structure to select the LOD appropriate for the current viewpoint [15], [21], [46]. Other methods replace distant geometry by their images (imposters) [31].

Although each of these methods provides significant rendering speed up, they must be used in combination to ensure that the number of polygons used in a view remain bounded even if the actual model is arbitrarily large. Too many highly detailed objects visible in the distance can slow down display. Similarly, too many hidden objects, even if viewed at low detail, can quickly increase complexity. One common approach to combine both visibility and detail reduction is to use an "estimate of the visibility" of an object to derive its required detail. Objects unlikely to be visible from the current viewpoint are drawn at reduced detail. El Sana et al. [14] use a density-based heuristic to compute a probability function to measure visibility. Objects with low probability of visibility are rendered at lower detail. Andújar et al. [3] similarly define "hardly visible sets" and reduce the detail of partially occluded objects by increasing the geometric error allowed for it. Others compute visibility from a region. Wang et al. [42] compute visibility by hierarchically subdividing rays originating from a region in space into beams. At the leaf levels, if a beam contains too many polygons, they prerender them and sample the color. At rendering time, they simply select the appropriate beams to process. The algorithm by Law and Tan [27] is similar in spirit. They precompute conservative visibility from regions of space and use this information to guide the occlusion preserving LOD refinement during rendering. These algorithms process a large amount of geometry at the rendering time and hence incur too much overhead to be practical for massive models.

Funkhouser et al. [16] presented one of the earliest comprehensive frameworks integrating visibility and detail control into a single system. They only considered axis aligned cells for region-based visibility precomputation, however. This was suited for many building models. Due to smaller model sizes, they did not need sophisticated layout or compression algorithms. An important characteristic of this and most other recent algorithms is that they precompute visibility for each view cell as well as a set of LODs for the objects in the model and later determine the appropriate LOD for the potentially visible set at rendering time.

Aliaga et al. [1] present MMR, a comprehensive rendering system that integrates the different methods. In their system, the different LODs, visibility, and image imposters are separately processed in a pipeline. Viewpoint-based visibility computation is performed at rendering time and is expensive. Recent works [4], [19], [41] speed up display by employing multiple processors and machines and by accepting increased latency and increased error, sometimes 10–15 pixels large, to guarantee interactive speeds. Our goals are different.

We believe that, as models grow large, any significant per-viewpoint computation becomes prohibitive for interactive visualization on commodity workstations. As a result, one must either compromise on quality or complete much of the computation beforehand. The first approach is well explored in the literature. In the second approach, one could predictively compute region visibility for cells in the neighborhood of the viewpoint. Current region visibility algorithms are not able to provide results fast enough. One could argue that if the cells were large enough and the viewpoint changed slowly, one might be able to devise an algorithm to finish this precomputation in time. On the other hand, large cells imply large overestimates and result in too much geometry per frame. One would need to further reduce this set by using a per-viewpoint approach. We have decided to explore the extreme end of this spectrum. We do most of our computation in a preprocessing phase and thus allow the online rendering algorithm to perform work proportional only to the geometry visible from the current view-region, already reduced to an appropriate detail. In order to achieve a high degree of fidelity while maintaining a low working set of polygons online, the precomputation phase first determines the acceptable LOD for a view cell and then keeps reducing its visible set until this working set is small in size. If the visible set remains large, it refines the view-cell and reiterates. In contrast, previous works first compute or estimate visibility and then reduce the LOD as much as necessary to obtain a low working set. As a result, they are limited by either a large working set or low fidelity.

3 Overview

In this section, we provide an overview of our system as well as define some of the terms that will be used in the rest of the paper. Our system has two main parts—the *precomputation phase* and the *runtime system*. A conceptual view of our system is shown in Fig 2.

In our system design, the *precomputation phase* does most of the work to keep the CPU load at a minimum during the walkthrough. We start by subdividing the view space into *view-cells* (or simply *cells*) using an octree-based partitioning scheme. The next step is to simplify the given 3D model and generate a set of discrete LODs. The granularity that the simplification algorithm operates on is at the level of topologically connected components of the model. We refer to these components as *objects* in the paper. Each object has a unique ID associated with it. The output of the simplification algorithm is LODs of all the objects in the model.

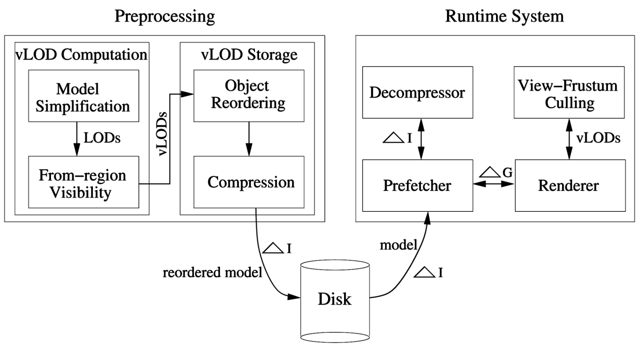


Fig. 2. Block diagram of the system architecture.

The next step is responsible for the generation of *vLODs*. A *vLOD*, from a given cell, refers to the visible portions of the original model at the right level of detail from any viewpoint (and view direction) inside the cell. The LOD for each object is chosen to conservatively maintain our one pixel error threshold as seen from all points in the cell. Based on this criteria for LOD selection, we determine the visibility information inside the cell using a from-region visibility algorithm. The granularity at which this information is computed can be different from that used for simplification, although, in our system, we do not distinguish between the two.

The next stage in our system pipeline is *vLOD storage*. The total size of the vLODs generated in the previous phase is usually much larger than the original model and is too large to be stored directly on disk. Fortunately, the coherence in the vLODs between adjacent cells allows significant compression. We compute the differences in vLOD information across neighboring cells and encode this information using the object IDs. We refer to this information as ΔI and the associated geometry as ΔG. The ΔI can be compressed using standard compression algorithms, but a clever reordering of the object IDs can achieve significantly better compression. Another benefit of the reordering algorithm is that it yields a layout of the model geometry on the disk that reduces disk latency. Given this permutation of the object IDs, we store the compressed ΔI and the permuted geometry on disk. This completes the precomputation phase.

The runtime part of our system is lightweight. It runs two threads: *renderer* and *prefetcher*. The renderer sends the current cell's vLOD to the graphics pipeline. The prefetcher loads, decompresses, and caches the ΔI and ΔG for each boundary of the current cell. When a new cell is entered, the renderer computes the vLOD for the new frame by updating the previous cell's vLOD with the corresponding boundary's ΔG.

4 vLOD Computation

The first portion of our preprocessing involves the computation of vLOD, our description of the visible geometry at the right level of detail for all places in the viewing space. To solve this problem, we decompose the viewing space into viewing cells. Both the visibility and the level of detail are held constant for all viewpoints within a cell and the quality guarantees are designed to be conservative for the entire range of the cell.

The relationship between visibility and the level of detail is somewhat complex. Adjusting the level of detail of the objects affects both the computational complexity and the output of a visibility algorithm. Similarly, changes in the visibility affect the bound on LOD error as well as the LOD number that may be used to achieve a given triangle count.

At a high level, our strategy for computing vLOD is as follows: For a given viewing cell, we compute a consistent level of detail for the entire scene to guarantee a high-fidelity rendering (i.e., with less than one screen pixel of surface deviation). Then, we compute the conservative from-region visibility for this cell. If the vLOD resulting from this visibility is larger than some threshold, we attempt to repeatedly reduce the vLOD size by shrinking the size of the cell.

Notice that our optimization for offline, from-region visibility is different for the approach used by online visibility computations. Such online systems are more constrained by processing times, so they typically make one quick pass at a visibility computation and then decrease the LOD fidelity until the target visible set size is reached. This produces qualitatively different results from our offline computation, which provides high-fidelity rendering, including correct (conservative) visibility. We next describe our simplification procedure before presenting the visibility computation algorithm.

Polygon Simplification. Before performing the simplification procedure, we partition the model into components we call *objects*. For architectural and CAD models, it is convenient to perform this partitioning by applying a connected component's algorithm and assigning each connected component an object ID. We perform an iterative edge collapse algorithm [21] to simplify each of these individual objects into a discrete LOD hierarchy, reducing the number of polygons by roughly a factor of two at each successive LOD.

Many such simplification algorithms are usable [30]. The main criterion for our system is that the algorithm should output a conservative bound on the maximum object-space surface deviation for each discrete LOD. We apply the simplest such metric, accumulating our error bound from the lengths of the collapsed edges, similar to Rossignac and Borrel [35].

For a given view-cell, we select an LOD for each part so that the screen-space surface deviation is less than a pixel. The ratio of the object-space error to screen-space error is computed by maximizing the projection of a unit vector at any point of the object [6] from any point in the view cell (we assume a maximum screen resolution for this computation). This LOD is consistent over the entire scene and guarantees high geometric fidelity when used from anywhere in the cell. Furthermore, this small screen-space deviation fits well with our desire to use simplified geometry during subsequent visibility computation.

5 Visibility Computation

In our system, we cull both polygons that are back-facing as well as those that are occluded from the current view-cell.

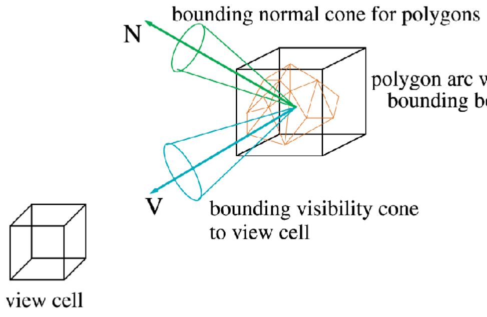

N
bounding normal cone for polygons
polygon arc with bounding box
V
bounding visibility cone to view cell
view cell

Fig. 3. Cell-based hierarchical back-face culling.

These computations are done entirely offline. We describe these algorithms next.

5.1 Hierarchical Back-Face Culling

Back-face culling is a simple, yet effective technique to prune out polygons that face away from the viewer. Instead of testing individual polygons, significant gains can be obtained if a collection of polygons can be classified as front or back-facing using a single test [26]. This test can be used as part of a hierarchical strategy for performing back-face culling. However, instead of performing the test from a single viewpoint, our test needs to determine their orientation from all viewpoints inside a single cell.

We start with a collection of polygons, its axis-aligned bounding box, B, and a view cell (see Fig. 3). We first compute a cone, $(\hat{\mathbf{N}}, \theta)$, bounding the normals of these polygons, where $\hat{\mathbf{N}}$ is the axis and θ is the half angle of the cone. We use a modified version [6] of an algorithm by Sederberg and Meyers [37] to compute this bounding cone, which may have its apex anywhere inside B. For the sake of argument, let this point be v.

Consider the set of all view rays starting at v and ending anywhere inside the view cell. These rays can be bounded by another cone whose apex is at v and that bounds the vectors from v to the vertices of the view cell. We call this cone the *visibility cone*, $(\hat{\mathbf{V}}_{\mathbf{v}}, \alpha_v)$. Fig. 3 shows the visibility cone.

Given these two cones, we can now test if any of the polygons at v is front or back-facing from anywhere inside the view cell as follows: Let us denote by $\Gamma_v = \cos^{-1} \hat{\mathbf{N}} \cdot \hat{\mathbf{V}}_{\mathbf{v}} + \alpha_v + \theta$ and $\gamma_v = \cos^{-1} \hat{\mathbf{N}} \cdot \hat{\mathbf{V}}_{\mathbf{v}} - \alpha_v - \theta$, the maximum and minimum possible angles between two vectors, one chosen to lie within the normal cone and the other chosen to lie within the visibility cone. Since v can lie anywhere inside B, we compute visibility cones from the eight corners of the bounding box B and update the extremal values of Γ_v and γ_v. Let the overall maxima and minima be Γ and γ, respectively.

If $\Gamma < \frac{\pi}{2}$, then any polygon inside B is *front-facing* with respect to the view cell. If $\gamma > \frac{\pi}{2}$, then any polygon inside B is *back-facing* with respect to the view cell. If neither of these conditions are satisfied, the result is inconclusive and we partition the set of polygons into two halves and proceed recursively. At the end of the recursion, we have a set of polygons classified as *front-facing*, *back-facing*, or *neither*. We discard the set if it is back-facing.

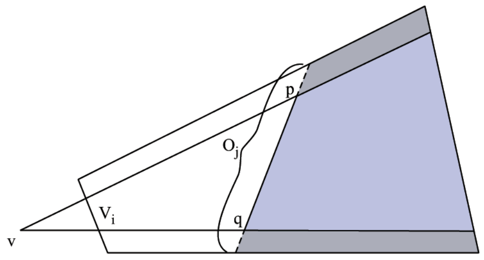


Fig. 4. View-cell visibility computed from a point.

5.2 Occlusion Culling

Most 3D cell-visibility computation techniques are simply impractical for large models. We exploit the significant hardware rendering speeds available on the graphics cards using an effective, yet practical algorithm. It is fast enough to potentially use all polygons as occluders. On the other hand, small polygons (relative to the cell size) shrink to null. Hence it combines adjacent polygons to form larger occluders. Furthermore, it performs occluder fusion, detecting if an object is hidden jointly by a combination of occluders in many cases. It parallelizes well and is highly scalable.

Given a set of polygons P, occlusion culling is the process of finding the subset H^v that is completely hidden from a view point v by a set of polygons $\{O|O \subset P\}$ [10], [25], [28], [43], [44], [47]. We call O the set of *occluders*. Given a view-cell V, it is also possible to compute H^V, the set of polygons hidden from every viewpoint $v \in V$. Clearly, $H^V \subset H^v$ and, hence, all polygons in H^V continue to be invisible from v. In our system, view-cell-based visibility precomputation is used to speed up rendering.

Recall that our goal is to eliminate polygons in the shadow of other polygons when viewed from the cell. If we could use the hardware to clip against this umbra region, we would obtain an efficient algorithm. Unfortunately, the umbra is bounded by ruled quadratic surfaces with negative curvature [38] and is tough to compute precisely. On the other hand, we can find volumes completely contained in the umbra and bounded only by planes. That is our goal. To achieve this, we shrink the volume further to transform it into a frustum and thus reduce volume-shrinkage to occluder-shrinkage. This allows us to exploit the graphics hardware by simply drawing the shrunk occluder polygons.

Consider the 2D illustration in Fig. 4, with view-cell V_i and occluder O_j. If we construct the supporting planes [9], tangential to both V_i and O_j, we obtain a shadow area (light shaded area) contained in the umbra. Now, consider a point v contained within the supporting planes. If we draw planes passing through v and parallel to the supporting planes, we obtain a view frustum (dark shaded area) contained in the shadow. Every polygon hidden by pq, (thick line), when seen from v, is hence guaranteed to be hidden by O_j from every point on V_i. We call pq the shrunk version of O_j. The shrinking depends on the choice of point v. This shrinkage is different from that by Durand et al. [13] because we shrink occluders directly in the object space instead of the screen space and do not need to expand the occludees. It is also different from the shrinkage used by Wonka et al. [44]. Their shrinking is a function of the radius of the view-cell. Thus, long or thin view-cells cause over shrinking. Furthermore, they shrink in all directions in 3D, requiring the occluders to have large interiors. Décoret et al. [10] reduce the overestimation for 2.5D models by using the Mikowski difference of occluders and the view-cell, but it is difficult to generalize in 3D. In contrast, we allow 3D occluders but need to shrink only in the planes of the triangles of an occluder and do not even require them to have an interior. Thus, we can use more polygons as occluders and do not suffer when cells have large aspect ratios. We accept a single triangle or a group of connected triangles as occluders.

We partition occluders into clusters based on their direction from v to simplify rendering. To compute visibility with respect to each cluster, we first construct the shadow frustum for each occluder (see Fig. 5). We next find a projection point v contained in all frusta of the cluster. We then shrink each occluder in the cluster by its reduced-shadow planes: planes parallel to shadow planes and passing through v. Occlusion behind shrunk occluders can be efficiently determined using the NVIDIA Occlusion Query OpenGL extension. We render the shrunk occluders first. We then draw the occludees with depth-buffer write disabled and the Occlusion Query extension enabled. The query returns a zero for occludees not visible. The others form a *visible list*. This visible list is progressively reduced by rendering against each occluder cluster. We are able to compute the visibility of an object from a view-cell with respect to an occluder cluster by drawing all polygons of the object and checking if any pixel was written. Assigning a single occlusion query ID to each object allows us to process all objects in a single pass.

For multiple occluders, O_i, note that v must lie in the shadow frusta of all occluders. We find the best location for v in

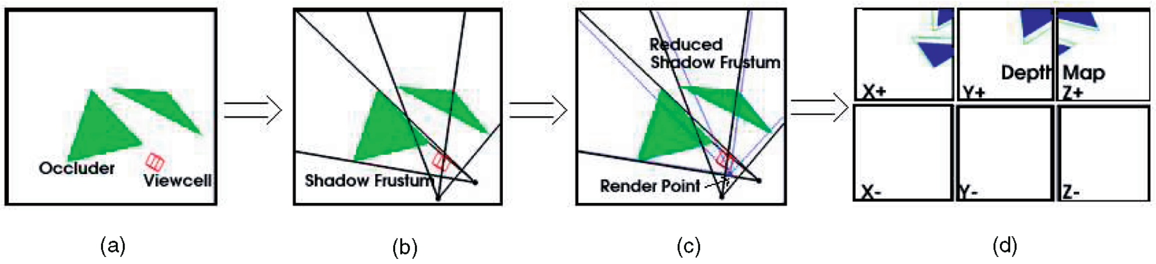


Fig. 5. 3D Visibility Pipeline: (a) Displays two triangle occluders with a viewcell. (b) Shows their shadow frusta. (c) Marks the projection point and the shrunk shadow frusta. (d) Visualizes the depth map from the projection point for six orthogonal directions.

TABLE 1
Properties of Some Recent From-Region Visibility Algorithms

Algorithm	Environment Type	Occluder Type	Cell Time	Test Machine	Model Size
Durand et al. [13]	3D	Polygons	20 seconds	SGI Onyx2 Inf Reality 200MHz R10K	6M
Schaufler et al. [36]	3D	Volumes	16 sec	PentiumII 400MHz	.6M
Wonka et al. [44]	2.5D	Extruded line segments	0.4 sec	PentiumIII 650MHz	7.9M
Koltun et al. [25]	2.5D	Height fields	24 sec	PentiumIII 750MHz	1.7M
Our algorithm	3D	General polygons	1 sec	AMD Athlon 1.53GHz	13M

the intersection of the frusta by maximizing the sum of volumes of all shrunk frusta. We refer to v as the *projection point*.

Computing the projection point. Having computed the shadow frustum for each of the occluders, we proceed to find the optimal projection point from which the shrunk shadow frusta determine conservative visibility. We formulate this as a convex quadratic optimization problem. Ideally, the projection point has the following properties:

1. For each occluder, O_j, the projection point must lie inside its shadow frustum.
2. The shrunk shadow frusta must maximize the amount of geometry lying completely inside them.

We relax constraint 2 to find the point that maximizes the sum of the volumes of the shrunk frusta. It can be shown that the constraint space under this formulation is convex with a quadratic objective function that does not have a unique maxima. Hence, we derive a dual objective function that is related to the sum of volumes

$$\Sigma_j \langle (v - p_j), \hat{n_j} \rangle^2,$$

where p_j is the optimal projection point for occluder O_j, $\hat{n_j}$ is the normal to planar occluder, and $\langle , \rangle$ denotes inner product. For nonplanar occluders, $\hat{n_j}$ is the average of normals. It can be shown [6] that this objective function minimizes the total reduction in the sum of volumes of the shrunk frusta. Intuitively, if the chosen point v is not at p_j, the shrinkage is minimized if v is only moved away from p_j along $\hat{n_j}$ or $-\hat{n_j}$. Recall that v must also satisfy the (linear) constraints imposed by property 1 above. This is the special case of a convex program called *Second-Order Cone Program* for which efficient interior point algorithms are known [29].

Using the above optimization decreases the visible set of most of our view-cells by 5-10 percent (over using a randomly selected point). The average time for solving the optimization problem is around 0.02 seconds.

Once v is determined, each occluder O_j is clipped to the frustum rooted at v with planes parallel to the supporting planes for O_j (or its umbra).

Note that rendering from the projection point using the graphics hardware need not be conservative. A pixel may only be partially covered by the occluders, but may yet be assigned their depth value causing occludees to be mis-classified as hidden. We perform conservative rendering by using alpha cut-off. We enable blending such that partially covered pixels result in an alpha value of less than one and, hence, are not updated with the depth value of the rendered occluders.

We compare some properties of a few recent from-region visibility algorithms in Table 1. The results quoted in the table are necessarily incomplete and approximate, but help provide a rough idea of the recent state of from-region visibility computation. We list both 2.5D and 3D visibility algorithms and any restrictions on the occluder type. Although Durand et al. [13] also present a single-plane projection variant that is more efficient than the one quoted in the table, we skip that data-point as that variant is well suited only for convex occluding polygons. The algorithm by Schaufler et al. [36] is highly memory intensive and is impractical for much larger models. Cell Time (the time for visibility computation per region) is quoted from the paper on the largest model (Model Size column) demonstrated in the respective papers. Naturally, these numbers are for different models and use different computers, but still are useful for comparison. These papers do not provide a consistent notion of the percentage of hidden geometry not culled (overestimation) and, hence, that statistic is not included in the table.

6 Data Storage

Although the size of vLOD geometry for each cell is only a small fraction of the total model size, it is impractical to store the vLOD of each cell separately on the disk due to the ensuing severe replication. For example, all cells of the power plant model in our experiment would require over 250 gigabytes. (The model itself is less than one gigabyte, including the LODs.) Furthermore, the disk bandwidth required to read each cell's vLOD would be beyond the capability of current disks.

In most cases, the vLODs of two adjacent cells exhibit high coherence and have mostly common geometry. $|\Delta G|$, the cardinality of the set difference between adjacent cells, is much smaller (of the order of 1 percent) than the full geometry. ΔG consists of two components, a list of geometry to be added, $\Delta^+ G$, and a list of geometry to be deleted, $\Delta^- G$.

We store a list of IDs, ΔI, for the geometry in ΔG for each boundary. This is conceptually similar to Durand et al. [13], but they store all the geometry in memory. This is not possible for larger models and ΔI as well as ΔG must be be fetched from the disk. We have devised a disk layout scheme that reduces the number of disk accesses by storing together on the disk objects that tend to be fetched together. The storage is still prohibitive, however, for large models with multiple LODs and, hence, further processing is necessary.

Sufficiently compressing the vLODs further [34] is a tough problem. We present in this section an algorithm that drastically reduces the storage requirement as well as the disk bandwidth requirement. Even more importantly, the decompression algorithm is extremely fast and does not impose any noticeable overhead at rendering time. For both disk layout and compression, we need the objects that are likely to be fetched together to have IDs close together. We solve this reordering problem next.

6.1 Object Reordering

For our application, the ideal layout is the one that minimizes a cost function that depends on the latency in disk reads. One formulation of this problem is to reorder the objects on the disk (or generate a permutation) so that objects fetched together always appear contiguously on the disk. This, in fact, is rarely possible without replicating objects on the disk. Funkhouser et al. [16] solved this problem by simply loading geometry in large clusters and keeping those clusters together on the disk. While increasing disk I/O granularity, this solution does not address coherence between clusters. Instead, we provide a finer grained solution that reduces the number of disk seeks required to fetch any desired object set. Solving this also improves the compression ratios we can achieve later: A simple run-length encoding scheme can now perform quite well.

Let us construct a binary matrix where the rows correspond to the face boundaries between adjacent cells and the columns to the object IDs. Consider a single row of the above matrix, viewed as a sequence of 0s and 1s. The number of disk seeks required is precisely the number of maximal consecutive sequences of 1s, also called the number of *runs* in the sequence. The number of runs can be changed by reordering the columns. An appropriate reordering, by reducing the number of runs, will reduce the number of seeks required to retrieve the data from this row. For a fixed row, it is always possible to reorder the columns to yield only one run; this may not be the best solution for other rows.

The reordering problem can therefore be restated as: Find a reordering of the object IDs (columns) so that the total number of runs (summed over all rows of the matrix) is minimized. This problem is closely related to the well-known *traveling salesman problem* (TSP) [23].[1]

For two binary vectors u, v of the same length, the *Hamming distance* between u and v is defined as the number of bit positions in which they differ. It is easy to see that, if we consider each *column* of the matrix to be a binary vector and compute the traveling salesman tour of minimum distance (under the Hamming metric) between these vectors, the resulting permutation also minimizes the total number of runs [2]. While the matrix reordering problem is NP-hard (by reducing the Hamiltonian path to it [24]), its connection to TSP suggests an algorithmic strategy. We can use fast heuristics for solving TSP [23] to obtain a reordering of high quality. The challenge in our case is dealing with the huge size of the matrix; Meneveaux et al. [32] used a number of heuristics to solve TSP to speed up radiosity computation, but those do not work well with the much larger matrix sizes in our problem.

There are two sources of complexity in using TSP methods for the reordering problem. As mentioned above, the size of the matrix precludes the use of most standard TSP methods. Second, and less obvious, the large number of rows in the matrix yields points in a very high-dimensional Hamming space, which makes even a single distance computation an expensive operation.

The best implementation for computing a near-optimal TSP tour of points in a metric space uses a *seed tour* that is then refined by heuristics that break the tour and recombine it to achieve a local improvement [23]. This seed tour is computed using nearest neighbors; start at some point, pick its nearest neighbor, and then repeat, choosing, for each point, its closest neighbor that is not already in the tour (the nearest *unvisited* neighbor). In practice, the algorithm constructs a list of the k nearest neighbors for each point in a preprcessing phase: If k is chosen suitably, then, for all but a small fraction of points, the nearest unvisited neigbour will be in this list and a general pairwise nearest neighbor query will be unnecessary. The design goal of our algorithm is thus two-fold: Find a small number of *representative neighbors* of a given point in the Hamming space (i.e., a column in the matrix) and use these to determine a viable tour that can be optimized further. We will use sampling to achieve these goals; the algorithm is described below. The constant k will be fixed later.

1. For each point, *sample* a fixed number of points at random and record the distances between this point and these samples. Sort this set and store the smallest k of them.
2. Initialize a TSP tour to be the first point.
3. While the TSP tour is not complete,
   a. Find the nearest unvisited neighbor among the samples for the current endpoint of tour.
   b. If no unvisited neighbor exists, pick any unvisited point and append that to the tour, using a dummy large cost for this edge.
   c. Perform local edge flips on the current tour for further optimization.

Nearest neighbor-based methods work well for Euclidean instances of TSP, giving answers within 25 percent of the optimal solution [23]. On smaller instances of the cell visibility data (for which we could compute the optimal permutation), we found that nearest neighbor heuristics are within a few percent of the optimal solution and using sampled data only decreases the quality slightly.

For Euclidean metrics, computing such nearest neighbors is relatively easy using data structures like *kd*-trees [5], but, for the Hamming metric, these strategies do not work. Thus, sampling is crucial to our algorithm to reduce the size of the input (and its dimensionality). A closer look at the data reveals the following observation, illustrated in Fig. 6. On average, if we determine the nearest neighbor of a point and consider the set of points within distance twice that of the nearest neighbor, then this set is extremely large, approaching a constant fraction of the entire point set. Moreover, although the maximum distance between a random point and its furthest point can be large, most of

1. It is interesting to note that a similar problem arises in reordering the computation of hierarchical radiosity of clusters of geometry to increase coherence [39], mining large graph structures, and association rule mining [24].

the points lie reasonably close to it (in terms of the ratio to the nearest neighbor distance). This suggests (and has been born out by preliminary experiments) that standard approaches for doing approximate Hamming metric nearest neighbor computations [8] are likely to perform badly because even a factor-two approximation to the nearest neighbor contains a large set of candidates to consider, driving the cost of determining nearest neighbors to $O(n)$ per point (and $O(n^2)$ overall). However, this anomalous behavior has a positive side; since a constant fraction of points lie within this ball, it implies that a constant-sized sample of points will contain at least one point within this ball with constant probability. Even more striking is the fact that a *single* sample set will suffice to provide near neighbors for *all* point sets. Formally, let $0 < \alpha < 1$ be the fraction of points lying in the ball of radius $2r_p$ around a point p, where r_p is the distance from p to its nearest neighbor. Then, with probability $(1 - 1/n)$, a sample of size $c \log n/\alpha$ will contain at least one point in this ball. In fact, there exists a constant c' such that a sample of size $c' \log n/\alpha$ points will contain a point within the ball of *each* point in the set. For experiments with our data, α is approximately 0.25, the sample size for one point $c' \log n/\alpha$ was 16, and k was chosen as 700.

This observation yields an approximation heuristic that runs in time $O(n \log n)$ instead of $O(n^2)$. We will not discuss this in detail here, but the above sampling can be designed to work in a disk-sensitive fashion so that it can be implemented on a system with small memory. This sampling strategy, along with providing fast access to near neighbors for a given point, also scales linearly with input size and is thus very well-suited for the large models we address.

We performed experiments on a 400,000 $\times$ 350,000 *power plant* visibility matrix and a 790,000 $\times$ 700,000 *city model* matrix (here, columns correspond to the input size of the TSP and rows indicate the dimension of the problem). In both cases, the average sparcity of each column was about 1,300, though the variance of the distribution was much higher in the second example. For the power plant example, we generated 700 near neighbors for each point from a sample of size 11,200 in slightly under 3 hours. For the city model example, we generated 700 near neighbors from a sample size of 22,400 in 7 hours. Computing the TSP tour itself is very fast: It rarely took more than 10-15 minutes for any input.

The total number of runs as a result of the TSP tour was found to be 64 percent fewer than a random permutation for the power plant model and 52 percent fewer for the city model. This improves read-speed and compression, as shown in the results section.

6.2 ΔI Compression

There has been recent research in compressing precomputed visibility information [34], [18]. Panne and Stewart [34], for example, represent the visibility information in a table of cell bit-masks. Each row corresponds to a cell and stores 1 bit per polygon, which is set to one for those that are visible. Columns and rows are collapsed based on a similarity metric to generate compressed version of the table. Unfortunately, in our case, due to the large number of polygons, such bit-mask based encoding of vLODs [18], [34] is prohibitively expensive. In fact, even simply storing each ID in ΔI as an integer already produces files more than 6–7 times smaller than raw bit-masks, which compares well with the compression ratios of 4–7 obtained by [34]. Our methods produce much higher compression.

TABLE 2
Performance of our Compression Algorithm

	Run Length Delta offsets	2-bit Delimiter	LZW	Adaptive Huffman
Compression Ratio	9	3.8	2.7	2.5
Decompression time per ΔI	10μs	21 μs	38 μs	36 μs

Column 1 shows the compression of raw bit-masks after the offsets are run-length encoded to obtain a sequence of small numbers. Columns 2, 3, and 4 compare further compression of that sequence. So, overall compression of our algorithm is a factor of 34.2 (9×3.8).

To compress a ΔI, we start by sorting its IDs and then variable-length encoding the offsets between consecutive IDs. Due to our object reordering algorithm, the IDs in a Δ are near each other, if not consecutive. This ensures that the offsets are small. Furthermore, due to an abundance of consecutive runs of offsets equal to one, this is well-suited for run-length encoding. Thus, our compression reduces to pairs of small numbers: (O_i, R_i), O_i is the offset from the previous ID in the offset list and R_i is the number of consecutive 1s in the offset list.

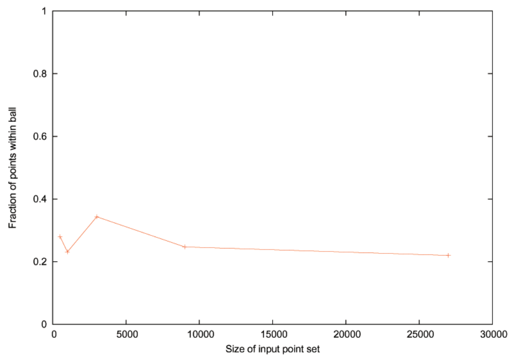
Techniques based on Huffman encoding usually work well with variable length encoding of text strings. However, literature on compressing a short list of small integers is scant. The frequencies of digits in such a list of numbers are uniformly distributed, thus affording Huffman coding little advantage over fixed length encoding. Furthermore, the overhead of storing a frequency table per Δ is significant. Statistical algorithms do not work well on small sequences either.

We use a two-bit delimiter-based technique for our compression. This achieves compression much more than gzip and Huffman encoding for our Δs and is much faster for both compression and decompression. Our compression and decompression algorithms make only a single pass over the input with no look-ahead and are described in Appendix A.

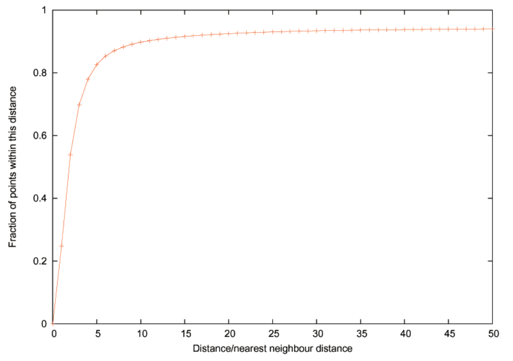
Due to the simplicity of the algorithm, our decompression is very fast and requires only a few microseconds per ΔI. We obtain better than 34x compression over the raw bit-masks. Our delimiter-based compressor generates output over 35 percent smaller than LZW compression of our run-length encoded data *and* our compressor as well as decompressor are nearly twice as fast. Table 2 shows the average compression over raw bit-mask and time to decompress each ΔI. To evaluate only the delimiter algorithm, we run LZW and Huffman on the pairs of small numbers after our run-length encoding. (The overall differences are even greater if we use LZW or Huffman algorithms directly on the original list of numbers.)

7 Runtime System

A major strength of our system is the simplicity of its run time display algorithm, as most of the hard work is finished beforehand. The renderer simply needs to reconstruct the



(a)



(b)

Fig. 6. Distance distribution statistics for a random point. In all cases, the numbers were obtained by averaging over multiple trials. (a) Density of points within a small ball around a random point. The y-axis plots the fraction of points within the ball and the x-axis shows the number of points in the point set. (b) Fraction of points contained in balls of increasing radius around a random point. The x-axis is the distance divided by the nearest neighbor distance.

vLOD of the cell the viewpoint enters. Assume the previous cell is C_{prev} and the current cell is C_{curr}. Also assume that $vLOD(C_{prev})$ is known in the steady state.

When the user moves to a new cell, we compute the sequence of face-boundaries, $\{B_i\}$, intersected by the path from C_{prev} to C_{curr}. In most cases, there is a single boundary crossed. We update the current vLOD for each boundary B_i as follows:

1. Fetch $\Delta I(B_i)$.
2. Decompress $\Delta^+ I(B_i)$.
3. Fetch geometry $\Delta^+ G(B_i)$ and append to vLOD.
4. Decompress $\Delta^- I(B_i)$.
5. Discard geometry $\Delta^- G(B_i)$.

In practice, we prefetch and cache ΔI and objects. Our current scheme is quite simple. The prefetcher thread generates a request for Δ of each cell adjacent to the current cell when the user enters a cell. It is also possible to configure the system to prefetch farther neighbors.

The cache manager sends a request to the disks for any data not already in cache. The cache uses the least recently used policy to replace objects or ΔIs when reading new data. When an object is added to the current vLOD, it is locked in cache so that it may not be discarded from the cache. When an object is deleted from vLOD, it is unlocked and enabled for replacement.

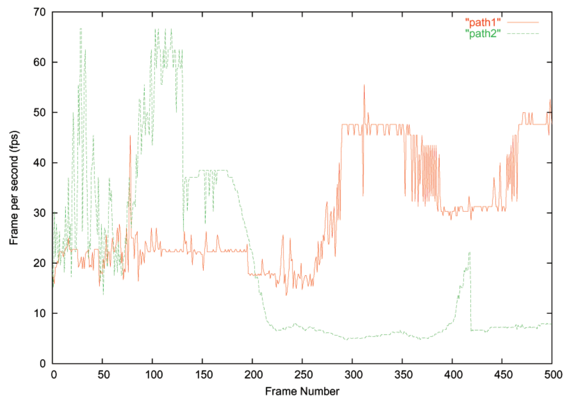


Fig. 7. Performance of system for two paths through the power plant model—path 1 and path 2.

Once the vLOD for the current cell is generated, it is passed to the view-frustum culler, which is not expensive as its input is only the objects in the vLOD. Objects found to intersect the view-frustum are finally sent to the rendering hardware.

8 Discussion of Results

We have implemented our preprocessing algorithm on a Beowulf cluster of AMD 1.53 GHz PCs with 1GB RAM each. Each machine is equipped with an NVIDIA GeForce 4 TI4200 graphics card which supports occlusion queries. The walkthrough system has been implemented on SGI-IRIX, Linux-based PC, and Windows-based PC platforms. We report results from a Pentium IV 2.8 GHz PC with 1GB RAM. The card on this PC is an NVIDIA GeForce FX 5800. We designated 512MB as the maximum cache size.

We have run our system on the UNC power plant model (13 million triangles, Fig. 1) and an artificial city model (34 million triangles, Fig. 11). We created the city model by stacking cubes and adding several CAD and scanned models to form a large city simply to verify the scalability of our system. The model contains several high detail areas and has a highly dynamic range of detail. For example, there are 50 cars on the street, each containing over 100,000 triangles. The cars are distributed in the city in groups of four to eight. Buildings also contain internal geometry. There is a high concentration of detailed geometry in the gallery, one of the city buildings. We maintained 10–20 frames for most of the city walkthrough. Since the UNC power plant model is well-known and publicly available, we focus our discussion primarily on that model.

Due to a large number of small polygons and open areas in the power plant model, this is a particularly tough case for occlusion culling. We maintain interactive rates on this model, as is evident in the example shown in Fig. 7. If the original polygons were sent to the model, the average frame rendering time is 3 seconds. Using only LOD (at 1 pixel error) improves this to nearly 1 second per frame.

In spite of our high rendering rate, the CPU load remains under 50 percent, all the time displaying high quality images. As shown in Fig. 1b and Fig. 1c, our LODs result in hardly any noticeable artifacts. Our visibility computation is strictly conservative and no polygon of the appropriate LOD that should be visible is discarded. We chose to accept

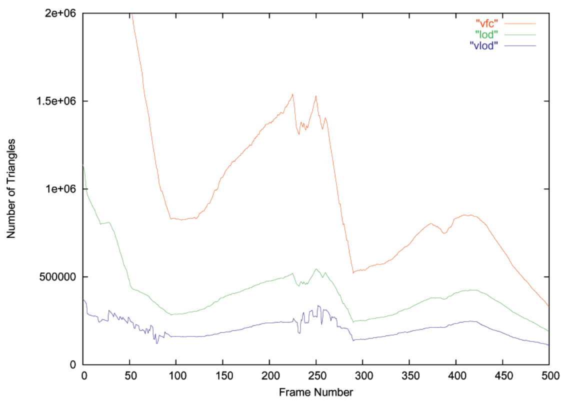


Fig. 8. Effectiveness of view frustum culling, view frustum culling plus LOD, and view frustum culling + vLOD along path 2.

only geometric LOD errors but no visibility error, as even small visibility errors are quite noticeable and distracting.

We have performed experiments to check the coherence between vLODs of adjacent cells taking several paths through the model. The average size of compressed ΔI is less than 750 bytes. The average number of polygons in each ΔG is less than 180,000.

In Figs. 9 and 10 we present some statistics for path 2, which was contrived to be a tough input for any walkthrough system. This path starts just outside the power plant and moves quickly to the most dense area of the model with a complex of pipes, looking directly at the pipes. Path 1, in contrast, starts with a bird's eye view of the whole power plant and the viewer flies in. The viewer moves fast, crossing 1-2 boundaries every second on average (cells are several inches to a few feet wide). No 3D region visibility algorithm can dynamically precompute the visibility from 2-3 cells and several of their neighbors per second for a 3D environment. That is why they are all precomputed in our approach.

We only try to maintain the Δ with respect to the six faces of an octree cube in the main memory. We use asynchronous reads to prefetch the relevant Δ from the disk. Fig. 12 shows the speed-up in disk-reads due to our layout algorithm. There was an almost 60 percent reduction in the number of disk reads and about 39 percent overall speed-up in rendering due to that.

In the precomputation stage, we partition an expanded bounding box of the model into nearly 500,000 view cells.

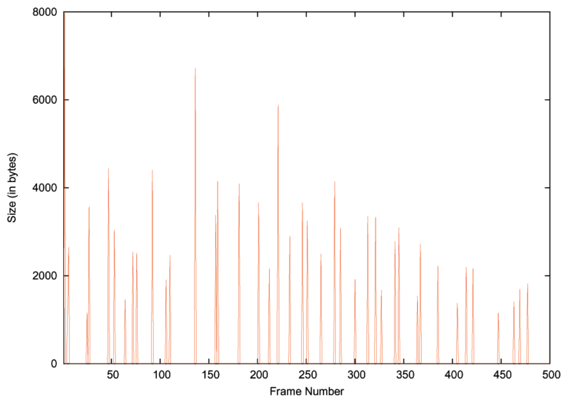


Fig. 9. Memory size of each ΔI along path 2 through power plant.

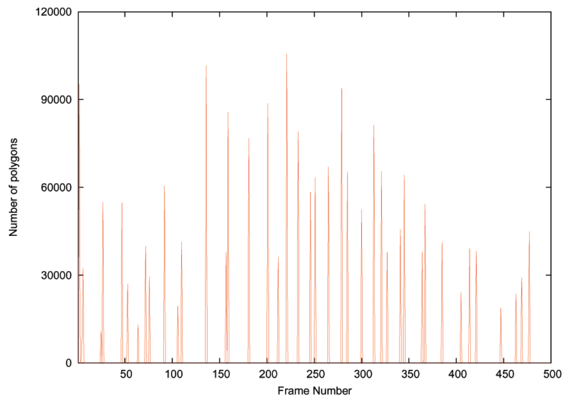


Fig. 10. Number of polygons in each ΔG along path 2 through power plant.

Most of these cells are concentrated in the area of complex pipes as the octree refinement goes deeper in those cells. The precomputation, running on a Beowulf cluster of 16 1.53GHz AMD Athlon-based Linux PCs, takes a cumulative total of 128 hours (the individual PCs taking from 8 to 10 hours). This produces a total of 7GB storage for ΔI. The time to compute vLOD is 0.4 to 2.2 seconds per view-cell. Of this, the time to shrink occluders is 0.16 seconds, while rendering and performing occlusion query takes about 0.2 to 2 seconds. Our Linux machines can render approximately 3-4 million triangles per second using vertex arrays and 50-250K occlusion queries per second.

In our hierarchical subdivision scheme, we employ two stopping criteria, which are the number of visible triangles and the level of the octree. About 30 percent of the view-cells that reach the lowest level octree require further

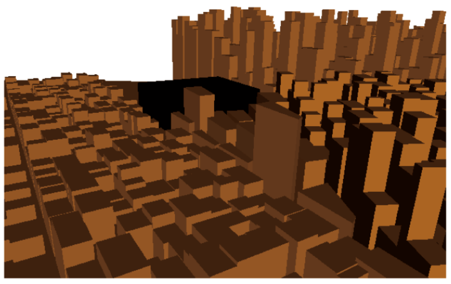


Fig. 11. A bird's eye view of the artificial city.

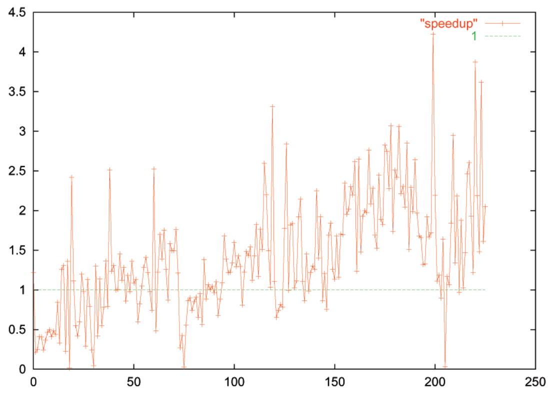


Fig. 12. Disk access speed-up due to object reordering during path 2 walkthrough.

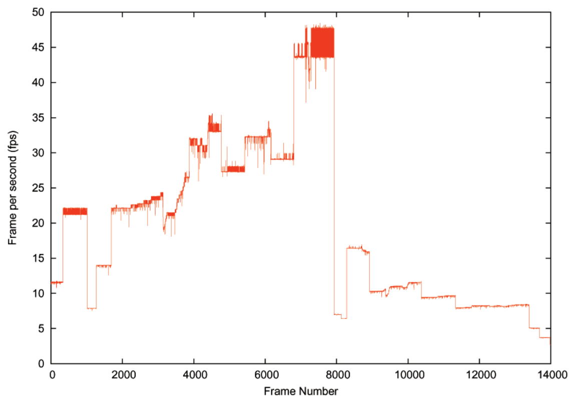


Fig. 13. Rendering performance of the system for the city model.

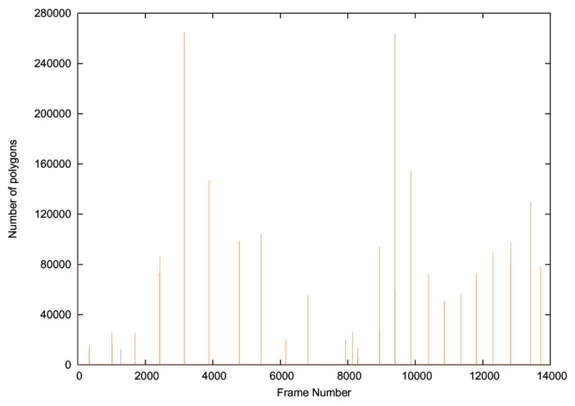


Fig. 15. Number of polygons in each ΔG for the city model.

subdivision. Their average vLOD size is about 1.5 times our triangle threshold.

We also perform an informal comparison of our cell visibility result versus estimated vLOD by taking the union of 125 sample points inside a view cell. The overestimation of our cell visibility algorithm is about two to 10 times the sampling result (which is itself an underestimate of vLOD for the view-cell).

For path 2 through the UNC power plant, we had to cross 128 cells. For most of the boundary transitions, we fetch less than 1 kilobyte of compressed ID and less than 2 megabytes of geometry data. For the same path, we plot the number of triangles rendered by our system and the number rendered after view frustum culling alone (Fig. 8). The number of triangles rendered by our system is much less than the number intersecting the view frustum. About half of the reduction comes from the use of high-fidelity LODs and the rest comes from discarding hidden geometry. On this path, we achieve an average rendering rate of nearly 30 frames per second with a screen space error of less than 1 pixel, leading to almost no visual difference from the full detail model.

Although the city model is larger (67 million polygons including LODs), it has only vertical walls used as occluders. In 2.5D our visibility precomputation is even more efficient [6]. The city was preprocessed in 14 (cumulative) hours, generating nearly 90,000 cells for a per-cell threshold of 0.5 million triangles. This results in 1.1 GB of storage for the compressed Δ. Fig. 13 shows the rendering performance of our system on the city model for a path that starts at one end, passes the densely populated region, and ends at the gallery. Figs. 14 and 15 exhibit coherence similar to the power plant model. Observe that the size of Δ does not grow as fast as the model size.

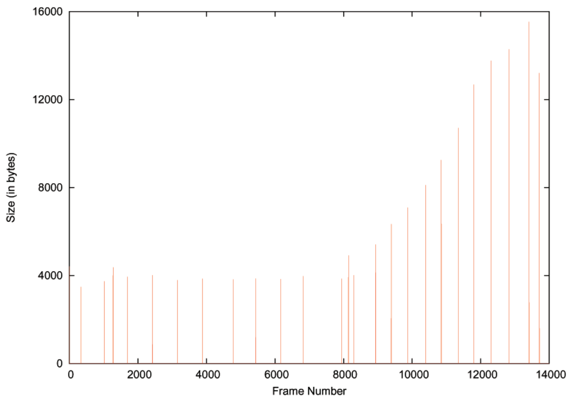


Fig. 14. Memory size of each ΔI for the city model.

9 Conclusion

We have presented a novel 3D visibility algorithm and a new technique to combine visibility computation and model simplification into a common framework. In fact, as more efficient simplification and visibility algorithms are developed, they can be easily incorporated into our system. This framework is particularly well-suited for models with high occlusion complexity and large spatial extent. We present algorithms to compress and organize the intermediate data on the disk that allow efficient fetching and decompression. Our implementation, even though lacking in many optimizations, is able to accurately and interactively display models with around 40 million polygons. This is significantly faster than has been achieved before. Our initial experimentation exhibits a high degree of scalability. With more accurate visibility precomputation and smarter partitioning, we believe our framework can eliminate graphics subsystem bottlenecks for a large class of visualization applications.

The memory footprint of our system is extremely low and we have been able to display large models even on commodity laptop computers. We can handle models larger than the main memory size as we only store the vLOD of the current view-cell and the Δ of its neighbors in memory at a time. Manageable hard disk storage overhead, low memory requirement, and interactive rendering speed make our framework an excellent vehicle for static model walkthrough and some computer games.

A number of optimizations can be performed on our existing system. The performance of the rendering algorithm is directly dependent on the size of the vLODs and this size can be reduced significantly by performing more aggressive occlusion culling and increasing the occluder set. Our current implementation is not able to bound the size of the vLOD for each cell; some cells still have large vLODs. Subdividing the view-cell further only increases the storage requirement without sufficient improvements in the vLOD size. We believe that a better spatial partitioning algorithm is needed to further improve the performance of this system. We currently allow no update of detail at the rendering time; this is an interesting area for future research. It would also be useful to allow some animation or other dynamic changes to the models. We believe small coherent changes can be handled in our framework.

Appendix A

Compression and decompression

A.1 Delimiter-based variable length compression

Our compressor requires a single pass over the input bit stream. In the following, we use italics to mark delimiter bits (after we recognize them as such). Note that the number 0 is never in the input. All other numbers are stripped off of their leading zeros and passed to the following compressor rules:

1. Add a two bit delimiter *11* between consecutive numbers in the sequence. This implies a string of *11*1 signifies the start of a new number at the last 1.
2. Replace instances of 11 in the input by 110 (note that delimiting *11* is always followed by a 1).
3. If a number ends in a 01 (possibly after applying rule 2), replace it by 011. This helps disambiguate the case of finding delimiters after a 1 (i.e., 1*11* from *11*1), as seen later.

A.2 Decompression

The decompression algorithm is also simple and performs a single pass over its input bit stream. It has the following steps:

1. If a 11*11*1 sequence is encountered in a string, append a 1 to the output and end the number, the last one in this sequence starts a new number. Note that this sequence could also be broken down as delimiter *11* followed by a new number. We mark this number as ambiguous and continue until we discover we have made a mistake.

   Justification: A 11110 is not allowed to appear at the start of a new number in the output. Note that the first 1 does belong to the new number but could not be alone as all instances of 1 get replaced by 11 in the compressed data by Rule 3. This happens if we have consumed an extra 11 in Step 1 before. We fix this by shifting right the ambiguous number last found by 1 bit. Note that the ambiguity only happens just before a string of 1s in the output and is discovered at the first non-1 number encoded in the data.
2. Otherwise, if a 111 is encountered, the current number is ended and a new one started at the last 1.
3. Otherwise, if a 110 is encountered, we discard the 0.
4. Otherwise, simply shift the next bit into the output.

References

[1] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha, "MMR: An Integrated Massive Model Rendering System Using Geometric and Image-Based Acceleration," *Proc. Symp. Interactive 3D Graphics,* pp. 101-106, 1999.

[2] F. Alizadeh, R.M. Karp, L.A. Newberg, and D.K. Weisser, "Physical Mapping of Chromosomes: A Combinatorial Problem in Molecular Biology," *Proc. ACM/SIAM Symp. Discrete Algorithms,* pp. 371-381, 1993.

[3] C. Andújar, C. Saona-Vázquez, I. Navazo, and P. Brunet, "Integrating Occlusion Culling and Levels of Detail through Hardly-Visible Sets," *Computer Graphics Forum,* vol. 19, no. 3, pp. C187-C194, Aug. 2000.

[4] W.V. Baxter, A. Sud, N. Govindaraju, and D. Manocha, "Gigawalk: Interactive Walkthrough of Complex Environments," Technical Report TR02-013, Univ. of North Carolina at Chapel Hill, 2002.

[5] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM,* vol. 18, no. 9, pp. 509-517, Sept., 1975.

[6] J. Chhugani, B. Purnomo, S. Krishnan, J. Cohen, and S. Kumar, "vLOD: A Scalable System for Interactive Walkthroughs of Very Large Virtual Environments," Technical Report JHU-CS-GL03-3, http://www.cs.jhu.edu/graphics/html/TR.html, Computer Science Dept., Johns Hopkins Univ., 2003.

[7] Y. Chrysanthou, D. Cohen-Or, and E. Zadicario, "Viewspace Partitioning of Densely Occluded Scenes," *Proc. 14th ACM Symp. Computational Geometry (SCG '98),* pp. 41-414, June 1998.

[8] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J.D. Ullman, and C. Yang, "Finding Interesting Associations without Support Pruning," *Proc. Int'l Conf. Data Eng.,* pp. 489-499, 2000.

[9] S. Coorg and S. Teller, "Real-time Occlusion Culling for Models with Large Occluders," *Proc. Symp. Interactive 3D Graphics,* pp. 83-90, 1997.

[10] X. Décoret, G. Debunne, and F. Sillion, "Erosion Based Visibility Preprocessing," *Proc. Eurographics Symp. Rendering,* 2003.

[11] F. Durand, G. Drettakis, and C. Puech, "The 3D Visibility Complex: A New Approach to the Problems of Accurate Visibility," *Proc. Eurographics Rendering Workshop 1996,* X. Pueyo and P. Schröder, eds., pp. 24-256, Aug. 1996.

[12] F. Durand, G. Drettakis, and C. Puech, "The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool," *Proc. ACM SIGGRAPH, Ann. Conf. Series (SIGGRAPH '97),* T. Whitted, ed., pp. 89-100, Aug. 1997.

[13] F. Durand, G. Drettakis, J. Thollot, and C. Puech, "Conservative Visibility Preprocessing Using Extended Projections," *Computer Graphics Proc., Ann. Conf. Series (SIGGRAPH '00),* pp. 239-248, 2000.

[14] J. El-Sana, N. Sokolovsky, and C. Silva, "Integrating Occlusion Culling with View-Dependent Rendering," *Proc. IEEE Visualization,* pp. 371-378, Aug. 2001.

[15] L. De Floriani, P. Magillo, and E. Puppo, "Building And Traversing a Surface at Variable Resolution," *Proc. IEEE Visualization '97,* Oct. 1997.

[16] T. Funkhouser, C. Séquin, and S. Teller, "Management of Large Amounts of Data in Interactive Walkthrough," *Proc. Symp. Interactive 3D Graphics,* pp. 11-20, 1992.

[17] Z. Gigus, J. Canny, and R. Seidel, "Efficiently Computing and Representing Aspect Graphs of Polyhedral Objects," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 13, no. 6, pp. 542-551, June 1991.

[18] C. Gotsman, O. Sudarsky, and J. Fayman, "Optimized Occlusion Culling Using Five-Dimensional Subdivision," *Computers and Graphics,* vol. 23, no. 5, pp. 645-654, 1999.

[19] N. Govindaraju, A. Sud, S. Yoon, and D. Manocha, "Parallel Occlusion Culling for Interactive Walkthroughs Using Multiple GPUs," Technical Report TR02-027, Univ. of North Carolina at Chapel Hill, 2002.

[20] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," *Proc. ACM SIGGRAPH,* pp. 231-238, 1993.

[21] H. Hoppe, "View Dependent Refinement of Progressive Meshes," *Proc. ACM SIGGRAPH,* pages 189-198, 1997.

[22] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang, "Accelerated Occlusion Culling Using Shadow Frusta," *Proc. Symp. Computational Geometry,* 1997.

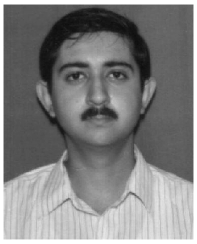
[23] D.S. Johnson, L.A. McGeoch, "The Traveling Salesman Problem: A Case Study in Local Optimization," *Local Search in Combinatorial Optimization,* E.H.L. Aarts and J.K. Lenstra, eds., pp. 215-310. London: John Wiley and Sons, 1997.

[24] D.S. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian, "Compressing Large Boolean Matrices Using Reordering Techniques," Technical Report TD-5X77W6, AT&T Technical Memorandum, March 2004.

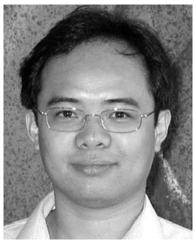
[25] V. Koltun, Y. Chrysanthou, and D. Cohen-Or, "Hardware-Accelerated From-Region Visibility Using a Dual Ray Space," *Proc. 12th Eurographics Rendering Workshop,* pp. 205-216, 2001.

[26] S. Kumar, D. Manocha, W. Garrett, and M. Lin, "Hierarchical Back-Face Culling," *Computers and Graphics,* vol. 23, no. 5, pp. 681-692, 1999.

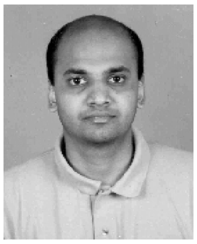
[27] F. Law and T. Tan, "Preprocessing Occlusion for Real-Time Selective Refinement," *Proc. Symp. Interactive 3D Graphics,* pp. 47-54, 1999.
[28] T. Leyvand, O. Sorkine, and D. Cohen-Or, "Ray Space Factorization for From-Region Visibility," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 595-604, 2003.
[29] M.S. Lobo, "Robust and Convex Optimization with Application in Finance," PhD thesis, Dept. of Electrical Eng., Stanford Univ., 2000.
[30] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner, *Level of Detail for 3D Graphics.* San Francisco: Morgan Kaufmann, 2002.
[31] P. Maciel and P. Shirley, "Visual Navigation of Large Environments Using Textured Clusters," *Proc. Symp. Interactive 3D Graphics,* pp. 95-102, 1995.
[32] D. Meneveaux, K. Bouatouch, and E. Maisel, "Memory Management Schemes for Radiosity Computation in Complex Environments," *Computer Graphics Int'l,* pp. 706-715, 1998.
[33] S. Nirenstein, E. Blake, and J. Gain, "Exact From Region Visibility Culling," *Proc. 15th Eurographics Rendering Workshop,* pp. 305-316, 2002.
[34] M.V. Panne and A.J. Stewart, "Efficient Compression Techniques for Precomputed Visibility," *Proc. 12th Eurographics Rendering Workshop,* pp. 305-316, 1999.
[35] J.R. Rossignac and P. Borrel, "Multi-Resolution 3D Approximations for Rendering Complex Scenes," *Geometric Modeling in Computer Graphics,* B. Falcidieno and T. Kunii, eds., pp. 455-465. Genova: Springer-Verlag, 1993.
[36] G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion, "Conservative Volumetric Visibility with Occluder Fusion," *Computer Graphics Proc., Ann. Conf. Series (SIGGRAPH '00),* Kurt Akeley, ed., pp. 229-238, 2000.
[37] T.W. Sederberg and R.J. Meyers, "Loop Detection in Surface Patch Intersections," *Computer Aided Geometric Design,* vol 5., pp. 161-171, 1988.
[38] S. Teller, "Computing the Antipenumbra of an Area Light Source," *Computer Graphics Proc., Ann. Conf. Series (SIGGRAPH '92),* pp. 139-48, 1992.
[39] S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan, "Partitioning and Ordering Large Radiosity Computations," *Proc. Ann. Conf. Series (SIGGRAPH '94),* pp. 443-450, 1994.
[40] S. Teller and C. Sequin, "Visibility Preprocessing for Interactive Walkthroughs," *ACM Computer Graphics (SIGGRAPH '97 Proc.),* vol. 25, no. 4, pp. 61-69, 1991.
[41] G. Varadhan and D. Manocha, "Out-Of-Core Rendering of Massive Geometric Environments," *Proc. IEEE Visualization,* 2002.
[42] Y. Wang, H. Bao, and Q. Peng, "Accelerated Walkthrough of Virtual Environments Based on Visibility Preprocessing and Simplification," *Computer Graphics Forum,* vol. 17, no. 3, pp. C187-C194, 1998.
[43] P. Wonka and D. Schmalstieg, "Occluder Shadows for Fast Walkthroughs of Urban Environments," *Computer Graphics Forum (Eurographics '99),* P. Brunet and R. Scopigno, eds., vol. 18, no. 3, pp. 51-60, 1999.
[44] P. Wonka, M. Wimmer, and D. Schmalstieg, "Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs," *Proc. 11th Eurographics Rendering Workshop,* June 2000.
[45] P. Wonka, M. Wimmer, and F. X. Sillion, "Instant Visibility," *Proc. EG Computer Graphics Forum 2001,* A. Chalmers and T.-M. Rhyne, eds., vol. 20, no. 3, pp. 411-421, 2001.
[46] J. Xia and A. Varshney, "A Dynamic View-Dependent Simplification for Polygonal Models," *Proc. IEEE Visualization,* pp. 327-334, 1996.
[47] H. Zhang, D. Manocha, T. Hudson, and K. Hoff, "Visibility Culling Using Hierarchical Occlusion Maps," *Proc. ACM SIGGRAPH,* pp. 77-88, 1997.



Jatin Chhugani received the BTech degree in computer science and engineering from the Indian Institute of Technology, Delhi, in 1999. He is a doctoral candidate in computer science at Johns Hopkins University, Baltimore, Maryland. His primary research interests are in real-time visualization, computational geometry, and image processing.



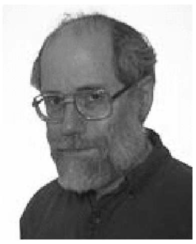
Budirijanto Purnomo received the BS and MS degrees in computer science from the Michigan Technological University, Houghton, Michigan, in 1999 and 2001. He is a doctoral candidate in computer science at the Johns Hopkins University, Baltimore, Maryland. His research interests are in computer graphics, visualization, and compiler optimization.



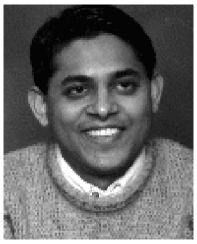
Shankar Krishnan received the BTech degree in computer science from the Indian Institute of Technology, Madras, and the MS and PhD degrees from the University of North Carolina, Chapel Hill. He is a member of the technical staff at AT&T Labs Research in the Information Visualization Department. His primary research interests include 3D computer graphics, computational geometry, and hardware-assisted geometric algorithms. He is a member of the ACM.

Jonathan Cohen received the PhD degree in computer science from the University of North Carolina at Chapel Hill in 1998 and the BA degree in computer science and music from Duke University in 1991. He is currently an assistant professor of computer science at Johns Hopkins University. He is most well-known for his research in polygonal mesh simplification. His general research interests are in the areas of interactive 3D visualization, geometric representation for 3D models, and parallel visualization systems.

Suresh Venkatasubramanian received the PhD degree from Stanford University in 1999 and has been at AT&T Labs-Research ever since. His research is primarily in computational geometry and algorithms and his recent work focuses on the use of graphics rendering devices to perform general purpose computations.



David S. Johnson received the PhD degree in mathematics from the Massachusetts Institute of Technology. He is head of the Algorithms and Optimization Department at AT&T Labs-Research. His research interests include the theory of NP-completeness, approximation algorithms for combinatorial problems, and the experimental analysis of algorithms. He is a fellow of the ACM and a member of SIAM, INFORMS, and the Mathematical Programming Society.



Subodh Kumar received the BTech degree in computer science from the Indian Institute of Technology, New Delhi, in 1991 and the MS and PhD degrees from the University of North Carolina, Chapel Hill, in 1993 and 1996. He is on the faculty of Johns Hopkins University. His primary research interests include Visualization, 3D computer graphics, and geometry processing. He is a member of the ACM and the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.