vLOD: A Scalable System for Interactive Walkthroughs of Very Large Virtual Environments

Jatin Chhugani
Johns Hopkins University
jatinch@cs.jhu.edu

Budirijanto Purnomo
Johns Hopkins University
bpurnomo@cs.jhu.edu

Shankar Krishnan
AT&T Labs - Research
krishnas@research.att.com

Jonathan Cohen
Johns Hopkins University
cohen@cs.jhu.edu

Subodh Kumar
Johns Hopkins University
subodh@cs.jhu.edu

March 19, 2003

Abstract

The problem of interactively rendering and navigating very large virtual environments has been well-studied in computer graphics. The two effective methods for efficient rendering of large polygonal models are to conservatively discard invisible geometry early in the pipeline and to remove unnecessary detail not discernible from the current view-point. There have been significant advances recently in both these areas independently but few comprehensive methods have been shown to derive maximum advantage of both methods at the same time. We present **vLOD**, a scalable system for performing interactive walkthroughs of very large geometric models on *commodity* graphics hardware. Our system performs work proportional to the *required* detail in *visible* geometry. We use a pre-computation phase to determine cell based visibility as well as level-of-detail. This pre-computation generates the geometry visible from a view cell at the right level of detail. We encode changes between neighboring cell's vLOD, which is not required to be memory resident. At rendering time, we incrementally re-construct the vLOD for the current view-cell and render it. Due to the small run-time overhead, we are able to display models with over tens of million polygons at interactive frame rates with less than *1 pixel error*.

1 Introduction

Computer modeling of data is essential in fields such as engineering, flight training, military exercises and medical and other simulations. Geometric models are used for

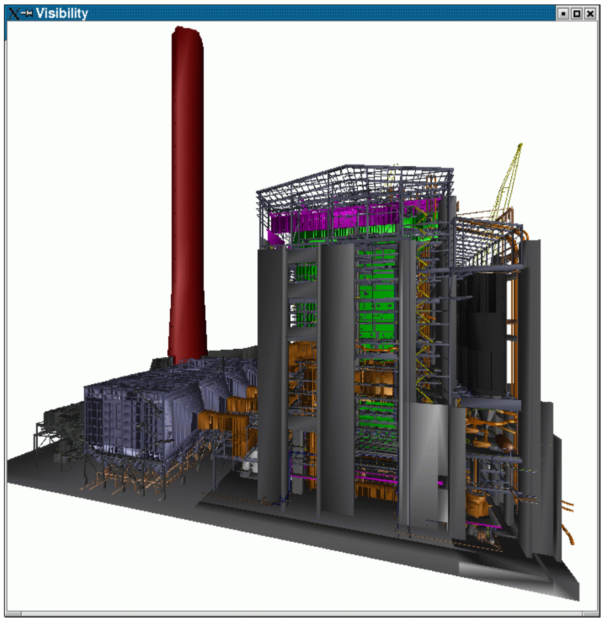


Figure 1: Powerplant Model

training, planning, design and other analyzes in these fields. Accurate modeling can reduce manufacturing cost and time to market or improve training of personnel who save lives. It is not surprising, then, that models have been quickly growing in complexity and richness. As a result, interactive walkthrough of these models is challenging.

It is true that the 3D model-display capability of widely available graphics hardware has been growing steadily. The sizes of models have grown faster, however. Models like ships, submarines and cities may require tens of million polygons or more to represent. Graphics workstations can barely display such models once in a few seconds. Interactive walkthrough requires the display of over 10–20 frames every second. Our goal is to enable *accurate* walkthrough of *large* models (even those that do not fit in the main memory) on *low resource* computers – all at the same time.

An important result of our work is to demonstrate that arbitrarily high graphics hardware rendering performance is not necessary to display models with arbitrarily high complexity. The basic premise of this argument is that display devices have limited resolution and that limits the detail that can be displayed. This implies that if the "right" set of triangles were displayed, no more than about one triangle per pixel would ever be necessary (assuming that sub-pixel accuracy is not required). An implication of this result is that the "right" algorithm can *accurately* display *arbitrarily sized* models at least ten times a second on a million-pixel screen using a graphics card that can display 10 million polygons per second. Of course, this algorithm would have to deliver to the graphics card only the visible objects at just the right detail; both are areas of active research.

Much of the related research has concentrated on visibility pre-computation, model pre-simplification, and image imposters. The first type of methods attempt to eliminate geometry not visible from the current view-point. Accurate visibility computation is a costly operation. Hence these algorithms conservatively compute a superset of visible geometry, which is passed on to the graphics hardware to determine true visibility. Simplification methods pre-compute a hierarchy of levels of detail (LODs) and at rendering time traverse a data structure to select the LOD appropriate from the current viewpoint. Imposter-based methods replace distant geometry by their images. All these methods provide significant rendering speed-ups. Unfortunately, there has been sparse research in methods to use these algorithms at the same time. [ESSS01] use a density based heuristic to compute a probability function to measure visibility. Objects with low probability of visibility are rendered at lower detail. [ASVNB00] similarly define "hardly visible sets" and reduce the detail of partially occluded objects by increasing the geometric error allowed for it. [WBP98] 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 pre-render them and sample the color. At rendering time they simply select the appropriate beams to render. The algorithm by [LT99] is similar in spirit. They pre-compute conservative visibility from regions of space and use this information to guide the occlusion preserving LOD refinement during rendering. These algorithms process a large fraction of the geometry at rendering time and suffer from too much rendering time overhead to be practical for massive models. Recently, an exciting new approach to scene-complexity independent rendering algorithm has been proposed [WFP$^+$01]. The main idea of this algorithm is to render only a small sample of the original set of triangles. This approach is only applicable to scenes with many sub-pixel triangles and does not work well on realistic scene environments with engineered models and high occlusion. Our method achieves similar input-size independence but is also applicable to more general scenes. We do this by incurring the cost of pre-processing when geometry relevant for different view-points are computed and explicitly stored, thus eliminating the need for any expensive data-structure traversal at rendering time.

Aliaga et al. [ACW$^+$99] present MMR, the first comprehensive rendering system that combines the different methods. In their system, the different LODs, visibility and image imposters are separately processed in a pipeline. The visibility computation is performed at the rendering time. One problem with the serialized pipeline approach of [ACW$^+$99] is the choice of order between stages. If one performs visibility first, one must devise a new algorithm to perform view-dependent adaptation of the level of detail only for the visible geometry. Traditional LOD methods assume a static model. Discretizing the geometry into bounding volumes and performing adaptation independently on a visible volume introduces artifacts like cracks. If, on the other hand, one adapts the detail first, cell based visibility pre-computation is not straightforward as the set of occluders and occludees both change dynamically. Recent work [WVBIM02, GSYM02, VM02] solves these problems by employing multiple processors and machines and tolerating increased latency and increased geometric error. Our goal is, instead, to focus on fidelity. We guarantee less than one pixel of error and allow the rendering time resource to be a single user level PC. In order for this method to be scalable we seek rendering times *independent* of the model size.

To avoid large computation at the rendering time, we must pre-compute some of it. Current techniques already pre-compute spatial partitions and simplifications hierarchies but still require rendering time computation that is dependent on the model size. Our key observation is that the actual model to be viewed from any position is, in fact, independent of the overall model size. Even if many objects are visible from a point, all the detail is not. In order to exploit it we must simply pre-compute all visibility and detail information beforehand and store it with the model. Then at rendering time, one only needs to fetch from a database the right (and small) set of triangles visible from the given viewpoint and display them. An added advantage of this approach is that it relieves the load on the CPU and enables it to perform any desired simulations on the visible part of model. On the other hand, our approach suffers from substantial pre-processing requirements. However, lifetime of many models is significantly longer than a single walkthrough. Once the pre-processing is complete and released, a multitude of users may be able to take advantage of a virtual walk through the model.

In this paper, we present a framework that derives the complete advantage of both simplification and visibility computation at the same time. While our implementation makes specific choices of simplification and visibility algorithms, in fact, many different geometric error-based simplification techniques and cell-based visibility algorithms can be "plugged into" our framework. We partition space into view-cells and, for each cell, pre-compute and store its *LOD*: the triangles visible from the cell at the appropriate detail. We show that neither the pre-processing time nor the disk storage requirement are prohibitive. There is usually high coherence exhibited in vLODs of adjacent cells. We compress and store the differences in vLOD between adjacent cells. At rendering time, we simply reconstruct the vLOD from these differences and display them.

Main Contributions: Our method scales well with increasing model size. We have been able to interactively render models with nearly 50 million triangle, most of which reside on the disk at each frame. Another primary consideration in our framework is to have a small memory footprint at rendering time, given the model sizes we deal with. While fast walkthrough is the main goal of this work, we have solved several component problems which are interesting in their own right and occur in a variety of applications. In particular, the important contributions of this paper are:

- **vLOD generation:** We provide an effective hardware accelerated algorithm to compute cell-based visibility and compute the appropriate LOD of the visible set.
- **Model Storage:** We provide a formulation of the disk layout problem and a heuristic to solve it. Customized disk layout of out of core information helps reduce the time taken to load data needed in any given frame. We also provide an algorithm for effective compression and efficient decompression of locations of objects on the disk. This problem has not been adequately addressed before. This ensures that vLODs are space efficient. Typically, our total vLOD storage requirements are similar to the original model size and thus we about double the total space requirements.
- **Fast Walkthrough system** Our vLOD-based algorithm is able to render large geometric data sets with single pixel geometric error at interactive speeds. We

are able to process models in excess of 50 million triangles, generate the vLODs, and render the geometry. Since we only maintain compressed vLODs for a few neighboring cells in memory at runtime, our rendering system has a small main memory requirement.

The primary limitation of the paper is the large pre-computation time. However, we do not need the pre-computation to be on a low-resource computer. Our pre-computation algorithm parallelizes very well and we have used a cluster of workstations performing pre-computations overnight. We believe this step can be speeded up significantly with further improvements in our algorithms. Our technique is best suited for environments containing high detail as well as high occlusion depth. It does not work well for environment with mostly small disconnected polygons.

A conceptual view of our system is shown in Fig. 2.

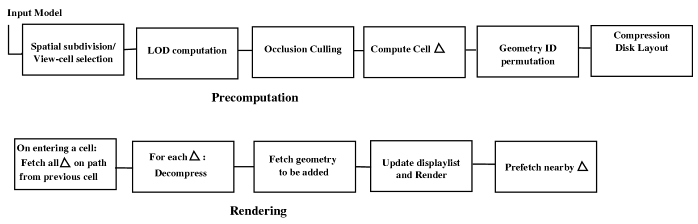


Figure 2: Block diagram of the system architecture

1.1 Other related work

Visibility computation and polygon simplification have been areas of substantial research recently. Both techniques provide significant rendering speedups.

Exact visibility computation from a given viewpoint is not very practical, hence most of existing research has focused on computing conservative approximations. Conservative algorithms retain *some* invisible geometry but do not cull *any* visible geometry. Visibility computation usually involves occlusion culling, back-face culling and view-frustum culling. Object-based visibility computation algorithms hierarchically compute the visibility for a group of triangles from the given viewpoint [GKM93, HMC$^+$97, CT97, ZMHH97, KMGL99]. On the other hand, cell-based visibility [TS91, CCOZ98, WWS00, DDTP00, SDDS00, WWS01] pre-computes conservative visibility from a region of space. At rendering time, objects visible from the view cell that the given viewpoint lies in are sent through the graphics pipeline to generate the correct display. Viewcell-based visibility computation has become popular recently due to its relative low cost at rendering time. An exact view cell visibility algorithm could be based on the computation of aspect graphs [GCS91] or visibility complex [DDP96, DDP97]. However, with size complexity of $O(n^9)$ and $O(n^4)$, respectively, and similar computation complexity, such approaches are infeasible for large models. The more practical approach is to subdivide space, say using an Octree, and pre-compute visibility from each cell in the space. Most current techniques choose a set of occluders appropriate for each view cell and cull out the hidden geometry. In addition to occlusion computation, hierarchical view-frustum and back-face computation algorithms [ZH97, KMGL99, JC01] have also been developed. Our method combines all three types of visibility computation into a view-cell based scheme. We use a variant of [WWS00] and [KMGL99] in our implementation.

Over the recent years, polygon simplification and LOD representation algorithms [Hop96, GH97, COM98, ME99] have also matured. While a small number of static levels of detail are useful if models are easily divided into objects, most environments require view-dependent refinement of detail [XV96, Hop97, FMP97], specially in the neighborhood of the viewer. The basic idea is to generate a set of simplification operations (e.g., edge collapse), which may be applied independent of each other. The rendering subsystem, then, chooses all applicable operations to perform that do not cause the screen-space error to exceed a given bound. We use a variant of [FMP98] in our implementation. [FMP98] generates a directed acyclic graph of nodes and arcs such that geometry is associated with the arcs and nodes with simplification operations. A cut through this graph represents the model at some level of detail.

1.2 Organization

The rest of this paper is organized as follows. The computation and management of vLOD is detailed in section 2. We discuss disk layout and geometry reordering in section 3. In sections 4 and 5 we provide details of the level of detail and visibility precomputation used in our system. Section 6 describes our implementation and results. We conclude in section 7

2 vLOD Management

In this section, we describe the computation and maintenance of vLODs in our framework. The basic concept of vLOD is quite simple:

- Hierarchically partition the viewing space.
- For each partition, or cell, adapt the level of detail so that the screen-space error when viewed from anywhere in the cell is bounded (using any geometric error-based simplification algorithm). Given, a screen-space error bound, we present in section 2.2 the procedure to compute the worst case object space error of all view-point in a convex cell.
- Obtain vLOD by discarding geometry that is guaranteed to not be visible from anywhere in the cell (using any cell-based visibility algorithm).
- Encode and compress the differences (Δ's) between vLOD of a cell and its neighboring cells.
- At rendering time, construct vLOD of the cell the viewer is in (using Δ's) and display it.

The differences are small on average, making the vLOD framework possible. To make it really practical, though, we must manage and encode the Δ's in such a way that storage requirements are reduced, and the time to reconstruct vLODs from them is very small. The strengths of vLOD framework lies in its simplicity and its ability to use any LOD adaptation and visibility computation algorithm. Our prototype implementation works well on commodity graphics hardware. It scales well and is able to interactively render models larger than available memory. vLOD also supports bounded screen-space error as well as guaranteed speed visualization (though, not both at the same time).

2.1 Spatial partitioning

Ideally, our view-space partitioning has three requirements:

- Visibility and detail of objects as seen from different points inside a single cell do no vary much. This ensures that we do not use much more geometry than are ideally needed for the given viewing parameters.
- The vLODs of adjacent cells do not vary by much. This ensures that the size of Δ's are small. Visibility sometimes changes drastically between cells. Hence, the worst case size of Δ can be large although the average size is small. (Section 2.3 describes how larger Δ's are compressed more efficiently.)
- The number of cells are small and their vLODs may be easily computed. This ensures small storage overhead and efficient preprocessing.

In our implementation, we use an octree-based partitioning scheme. We start with a uniform grid and refine it further. A cell, C, is too large and hence subdivided, if its vLOD(C) is larger than a user-specified value. Occasionally, subdividing these cells does not produce sufficient reduction in vLODs. In this case, we check the ratio of its vLOD and vLOD$_{min}(C)$, where vLOD$_{min}(C)$ is the smallest vLOD(C_i), for all subdivided cells, C_i, of C. If vLOD(C) is no more than twice vLOD$_{min}(C)$, we do not subdivide the cell C.

2.2 Object space to screen-space error transformation

In order to use view-dependent error in terms of the number of pixels, we need a function that maps the object-space error, $\delta_o(p)$, at point p to screen-space error δ_s. We define the *scaling factor*, $S(p)$, of p with respect to the view-cell such that $\delta_o(p) = \frac{\delta_s}{S(p)}$. The scaling factor is the maximum ratio between an infinitesimally small vector in the objects space and its projection in the screen space from any viewpoint in the view-cell. We compute $S(p)$ as follows:

The scaling factor for a point p with respect to the viewpoint v, $S_v(p) = \frac{(f+z)^2}{(f.L)}$ [CK01], where L is the distance of p from v, f is the focal length and $(f+z)$ is the projection length of the vector $\vec{vp}$ along the principle viewing direction.

For a view-cell, we find the maximum $S_v(p)$ over all viewpoints in the cell. This maxima, $S(p)$, is $L\cos^2(FOV)$, where FOV is the maximum field of view. The scaling factor of an object is the maximum $S(p)$ over all points p of the object.

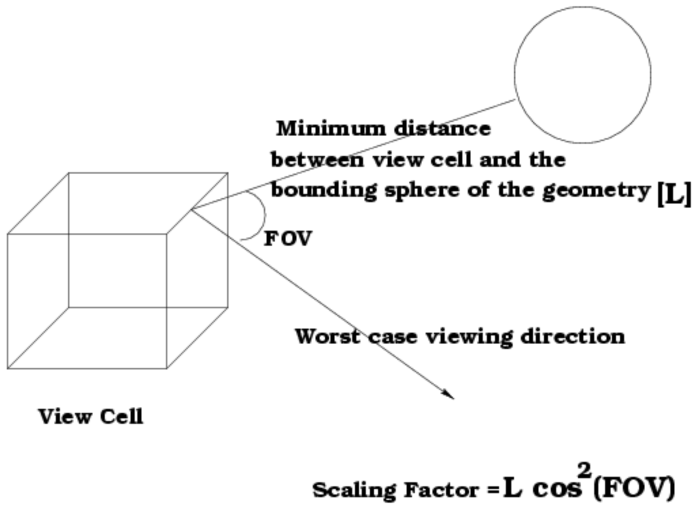

Minimum distance
between view cell and the
bounding sphere of the geometry [L]
FOV
Worst case viewing direction
View Cell
Scaling Factor = L cos^2(FOV)

Figure 3: Scaling factor of an arc from a view-cell

2.3 vLOD Construction and Cell Differences

In this section, we will describe our technique for computing and storing vLODs assuming a nominal visibility and simplification algorithm. The specific details of these algorithms will be described later in the paper, but for now, it suffices to assume that the simplification algorithm produces the objects at a detail so that the approximating error is less than a pixel when viewed from the current view-cell. We eliminate the objects which we determine to be invisible.

Most simplification algorithms are also capable of producing a sequence of operations (e.g., edge-collapse and vertex-merge) that can be efficiently encoded. For the sake of generality, we do not use these operations.

In most cases, the vLODs between two adjacent cells exhibit high coherence. Therefore, it is not necessary to store the complete vLOD set at every cell. Instead, we store the differences between their vLODS. We currently assume that the lists of objects in the LODs of two adjacent cells fits in the main memory. We sort them and compute the difference, which we compress and store. If cells C_i and C_j are adjacent (have a common face), we store the Δ associated with the boundary of C_i and C_j only once.

2.4 vLOD Δ Compression

There has been recent research in compressing pre-computed visibility information [PS99, GSF99]. In [PS99], for example, they represent the visibility information at each cell as a bit vector (one bit per polygon). 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, bit-mask based encoding of vLODs [PS99, GSF99] is prohibitively expensive. Our methods produce much higher compressions.

We have found that the number of IDs in a single Δ is usually small (less than 1% of the total number of IDs, even in models with many occlusion events). As a result, storing each ID appearing in Δ as an integer produces files of sizes more than 25 times smaller than raw bit-masks. We use variable-length encoding of offsets between consecutive IDs in a Δ. Techniques based on Huffman encoding usually work well with variable length encoding of text strings. However, literature on compressing a short and sorted list of small integers is scant. The frequencies of digits in a such a list of numbers are uniformly distributed, thus affording Huffman coding little advantage over fixed length encoding. Furthermore, the overhead of storing frequency table per Δ is significant. We use a two-bit a delimiter based technique for our compression. This achieves a compression even better than gzip and Huffman encoding for smaller δs and is much faster. For larger δs we us adaptive Huffman encoding.

Delimitor based Variable Length Encoding

Single pass compression:

1. Assume that 0 is never recorded and all numbers begin with a 1.
2. 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.
3. Replace instance of 11 in the input to 110 (note that delimiting *11* is always followed by a 1)
4. If a number ends in a 01 (possibly after applying rule 3), 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.

The decompression algorithm is also simple and performs a single pass over its input:

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.
2. 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 4. 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.
3. Otherwise, if a 111 is encountered the current number is ended and a new one started at the last 1.
4. Otherwise, if a 110 is encountered, we discard the 0

5. Otherwise, simply shift the next bit into the output

Due to the simplicity of the algorithm, our decompression is also very fast and requires only a few microseconds per Δ. If the input consists of long sequences of consecutive numbers we run-length encode Δs to obtain a further compression by a factor of 2. Our permutation algorithm (3) ensures that each Δ consists of several lists of consecutive values. If we store only the offset of each run's beginning from the previous run's end, and the current run's length. We obtain an even higher compression. In practice, we obtain compression ratio of greater than 100 over the raw bit masks, and output about 2 times smaller than direct LZW compression of the run-length encoded data.

3 Polygon Re-ordering for Compression and Reducing Disk Latency

Once all the Δs across cell transitions are computed, we need to lay them out on disk. The layout on disk can be chosen from a spectra of possible layouts whose extrema satisfy certain properties optimally. One extreme is to have all the geometry comprising individual Δs laid out contiguously, while the other extreme is to have the entire geometry laid out once. The former scheme is very efficient in terms of disk *seeks* and optimal in the amount of extra geometry that has to be read. However, all the Δs are laid out individually, the amount of the data replication is very high. The latter scheme is optimal in terms of the data replication factor, but the time required to read all the relevant information in a given Δ can be potentially very high. For our application, the ideal layout is one that minimizes a cost function whose terms include the replication factor and latency in disk reads.

We consider two variations of the optimization problem. In the first case, we consider the case of no replication with single reads allowed per Δ. This implies that the problem reduces to re-ordering the geometry (or generating a permutation) such that the polygons in each Δ appear contiguously in the order. This problem addresses the issue of minimizing disk seeks as well as the amount of extra geometry that is read in. If we construct a binary matrix where rows are indexed by cell transitions and columns by polygons, the above re-ordering problem is identical to the well-known *consecutive-ones* problem [BL76]. This problem and most of its variants are known to be NP-complete. In our case, the size of these matrices are related to the model size and can be in the order of 10^6.

We now describe a simple heuristic to achieve the consecutive-ones property. We start by giving some definitions to aid in the explanation.

Definition 3.1 Hamming Distance: *Given two d-dimensional bit vectors u and v, the Hamming distance $H(u,v)$ is defined as the minimum number of bit flips to go from u to v. In other words, $H(u,v) = \Sigma_{i=1}^{d}(u_i + v_i)\%2$, where u_i is the i^{th} bit in u.*

Definition 3.2 Run: *A* run *is a maximal consecutive sequence of 1's in a bit sequence. Given a bit vector u, the function $R(u)$ defines the number of runs in it. For example,* *the vector* $u = 0011100111010$ *has* $R(u) = 3$. *This can be extended to matrices by adding the number of runs in each row.*

Alizadeh et. al [AKNW93] showed the relationship to the matrix permutation and the travel salesman problem (TSP). They showed that given a matrix as defined above, the permutation induced by a minimum length Hamiltonian cycle (TSP) in the Hamming metric, minimizes the total number of its runs. Ideally, if the matrix has a *consecutive-ones property*, then each row will have exactly one run. Given the general infeasibility of such permutations, we are resorting to minimizing the total number of runs. Even though the TSP is an NP-complete problem, a number of efficient heuristics have been developed which produce close to optimal results in practice.

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 [JM97]. This seed tour is computed using nearest neighbours; start at some point, pick its nearest neighbour, and then repeat, choosing for each point its closest neighbour that is not already in the tour (the nearest *unvisited* neighbour). In practice, the algorithm constructs a list of the k nearest neighbours 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 nearest neighbour query will be unnecessary.

For Euclidean metrics, computing such nearest neighbours is relatively easy using data structures like *kd*-trees [Ben75]. However, for the Hamming metric, this is non-trivial. We implemented the greedy strategy to compute nearest neighbors on top of the traveling salesman algorithm for matrices of size upto 360000×27000. Once the seed tour is contructed, the algorithm goes through an iterative process of reducing the cost of the tour. We use $k = 20$ in our implementation. Table 1 shows the performance of various aspects of the algorithm. We can see that on an average the results of the TSP give a factor of 3-6 improvement in the number of runs over the original matrix.

Size	**kNN TIme**	**Seed Tour Time**	**Imp1**	**Imp500k**	**Time/Iter.**
500	2	0.7	3.1	6.28	0.005
1000	10	2.8	3.26	5.57	0.008
3000	162	31.1	2.84	4.56	0.015
9000	2400	584.1	2.78	4.25	0.025
27000	23393	7446.7	2.73	-	0.05

Table 1: Performance of the Hamming-Traveling Salesman Problem: **Size** refers to the columns in the matrix, **kNN TIme**: time to create list of k-nearest neighbors, **Seed Tour Time**: time to create greedy tour, **Imp1**: improvement ratio in the number of runs after first iteration, **Imp500k**: improvement ratio in cost after 500000 iterations, **Time/Iter.**: refers to the time taken per iteration. All times are in seconds measured on an SGI R10000 processor with 196 MHz. CPU and 7GB of main memory.

Note also that, in our case, the dimension of the Hamming space is very large (of the order of 10^5) and each point in this space contains only a few non-zero entries. Hence, the data representation is in the form of a sparse matrix (for each point, we maintain a list of the non-zero dimensions), and even the distance computation is expensive, being linear in the number of non-zero dimensions of the two points being considered, rather than constant for Euclidean metrics.

These facts make computing the desired tour a challenging problem. A closer look at the data reveals the following observation, illustrated in Figure 4. On average, if we determine the nearest neighbour of a point and consider the set of points within distance twice that of the nearest neighbour, 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 the points lie reasonably close to it (in terms of the ratio to the nearest neighbour distance). This suggests (and has been borne out by preliminary experiments) that standard approaches for doing approximate Hamming metric nearest neigbours computations [CDF$^+$00] are likely to perform badly, because even a factor-two approximation to the nearest neighbour contains a large set of candidates to consider, driving the cost of determining nearest neighbours to $O(n)$ per point (and $O(n^2)$ overall). However, this anomalous behaviour 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 the ball with constant probability. Even more striking is the fact that a *single* sample set will suffice to provide near neighbours 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 neighbour. Then with probability $1/n$, 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.

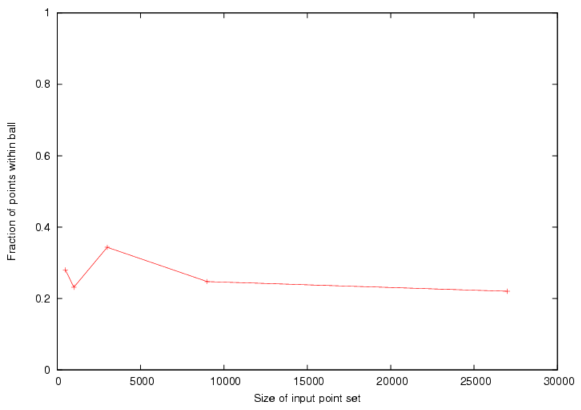
This observation yields an approximation heuristic that runs in time $O(n \log n)$ instead of $O(n^2)$. We will not discuss this in detail here, but the above sampling can be designed to work in a disk-sensitive fashion, so that it can be implemented on a system with small memory.

4 Polygon Simplification

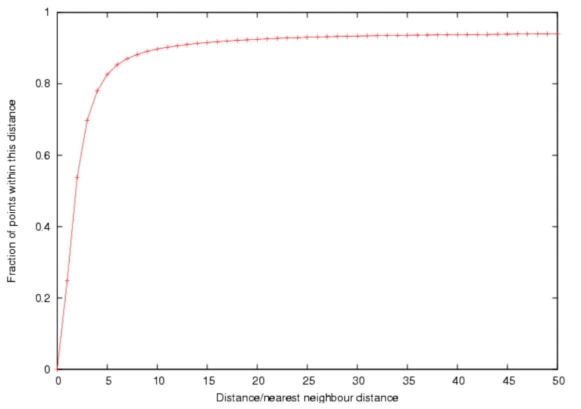
We have used a range of different simplification algorithms with our system, depending on the characteristics of the current model. For the city model, we applied an iterative edge-collapse algorithm with a projection-based error metric [CMO97]. This algorithm produces an MT data structure, capable of fine-grained view-dependent triangle refinement. For the powerplant model, we applied a more simple and robust half-edge collapse algorithm. The error metric in this case is a conservative bound accumulated from the lengths of the collapsed edges, similar to the bound provided by [RB93] and the first-pass error estimate made by [Gueziec96]. For this model, we output several discrete levels of detail rather than the more continuous level of detail encoded by the MT.

Whichever level of detail algorithm is employed, it is useful if it provides a guaranteed error bound for the geometric distance between the original and simplified surface (if it does not, a guaranteed error bound may be computed as a post-process [CRS98, ASCE02]. Without such an error bound, establishing a rigorous relationship between level of detail and visibility is impossible.

If we wish to preserve visibility conservatively in the presence of geometry level of



(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 neighbour distance

Figure 4: Distance distribution statistics for a random point. In all cases, the numbers were obtained by averaging over multiple trials.

detail, we should use an error threshold of less than one pixel of screen space deviation when we choose the levels of detail to use for a particular cell. In addition, we should use the same levels of detail during the visibility computation that we do during the rendering of cells. These conservative choices can provide some of the scalability benefit of levels of detail for large-scale models while delivering correct visibility at the given screen resolution.

The use of level of detail also has some effect on the optimization of polygon ordering for improved disk access. Each level of detail adds another polygon group to the set of geometry to be organized. However, because only one level of detail of each object may be visible from any cell, these additional polygon groups do not reduce increase the constraints on disk layout. In fact, these levels of details may be seen as a form of cheap replication which alleviates some of the constraints imposed by the many view cells.

5 Visibility Computations

Current cell-visibility computation techniques are simply impractical for large models. We exploit the significant hardware rendering speeds available on the graphics cards devise an effective, yet practical algorithm.

The main advantages of out visibility algorithm are that it can handle general scenes. The occluders can be collection of polygons. It is able to find occludees hidden jointly by multiple occluders. It parallelizes well. As a result it scales well with increasing model sizes.

We first briefly present an algorithm for environments restricted to 2.5D (vertical) occluders and then describe our general 3D occlusion computation algorithm and a hierarchical back-face culling algorithm.

Given a set of polygons $\{P_i\}$, occlusion culling is the process of finding the subset $\{H_i^v\}$, that are completely hidden from view-point v by a set of polygons $O_i, O \subset P$ [ZMHH97, WS99, WWS00, KCC01]. We call O the set of *occluders*. It is also possible to compute $\{H_i^V\}$, the set of polygon hidden from every viewpoint v in the set V (also called the view-cell) [ZMHH97, WS99, WWS00, KCC01]. Clearly, $\{H^V\} \subset \{H^v\}$, for all v in V and hence all polygons in $\{H^V\}$ continue to be invisible from v. In our system, view-cell based visibility pre-computation is used to speed up the rendering.

5.1 2.5D Occlusion culling

For some architectural scenes it is possible to choose occluder polygons that extend all the way to the floor. In other words for any point p on the occluder, the line pF_p is contained in the occluder, where F_p lies on the floor and pF_p is perpendicular to it. We call these 2.5D occluders.

Our 2.5D occlusion culling algorithm is based on the idea of *occluder shadows* by Wonka et al.[WS99]. This idea is based on shadow volumes [Cro77, HMC$^+$97]. They employ the graphics hardware to speed up the shadow test.

The algorithm of Wonka et al.[WS99] works as follows: Given a viewpoint v and a 2.5D occluder whose top edge has vertices a and b, they find a plane Π determined by v, a and b. Using this plane, they compute the vertices of a quad as $a+t(a-v)$, a, b and

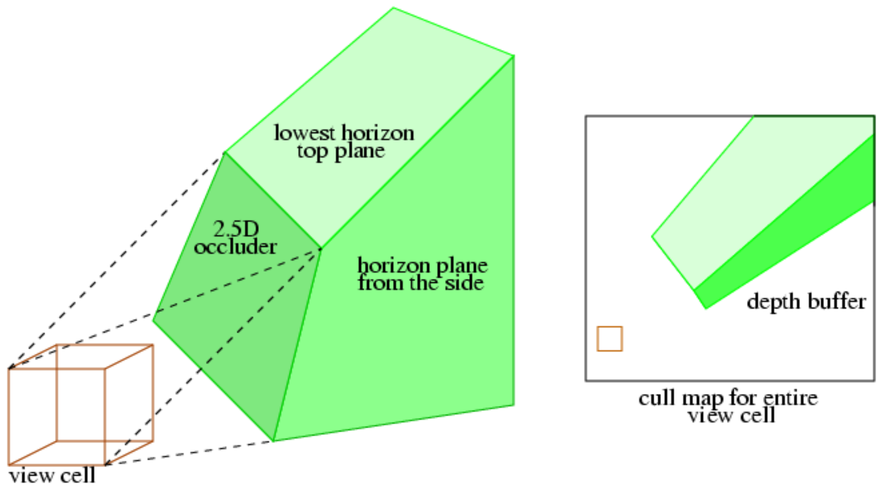


Figure 5: Occlusion culling using shadow volumes

$b + t(b - v)$ for a large value of the parameter t. These quads are computed for all the occluder polygons. For the special case of 2.5D occluders, the occluder shadows form a terrain. Using this fact, they render the quads orthographically from the top (calling the resulting image the cull map) and use the hardware z-buffer to fuse the shadows generated by various occluders. If the contents of the z-buffer are read back, occlusion tests can be performed very efficiently by simple depth checks. An object is occluded if its depth (when viewed from above) is larger than the corresponding entry in the cull map.

Wonka et al.[WWS00] later extended the idea of occluder shadows to perform view-cell based occlusion culling by subdividing the view-cell into small regions, and shrinking each occluder by an amount equal to the region's radius. The view-cell occlusion may then be conservatively estimated by occlusion from the center of the cell using the shrunk occluders. They then use a multi-pass rendering phase to compute conservative shadow volumes with respect to the original view-cell.

We use the idea of occluder shadows and cull maps as the basis of our algorithm, but eliminate the cost of subdivision.

Instead, our algorithm for finding cell-based shadow volumes computes a shadow frustum bounded by planes, which is guaranteed to be completely contained in the actual shadow-volume. This frustum is computed as an intersection of half-spaces. The bounding planes (called *horizon planes*) of each half-space passes through a single edge of the occluder and a vertex of the view cell such that the occluder and view cell are completely contained in the half-space. For each edge of the occluder, there is at least one vertex of the view cell that obeys this property (it can be proved by the feasibility of a simple linear program). We determine such horizon planes for each edge of the occluder. The intersection of the half-spaces determines a shadow volume that is guaranteed to lie completely inside the actual shadow because the shadow volume and (a bounding volume of) the view cell lie entirely on opposite sides of the occluder (see

Figure 5).

For the special case of 2.5D occluders, the shadow frusta that we compute also form a terrain. We employ the same strategy as [WS99] to draw these planes orthographically from the top and use the hardware z-buffer to speed it up. Once all the horizon planes are rendered, we read back the z- buffer for performing occlusion tests.

5.2 3D Occlusion culling

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 could obtain an efficient algorithm. Unfortunately, the umbra is bounded by ruled quadratic surfaces with negative curvature [Tel92] and is tough to precisely compute. In order to exploit the graphics hardware for occlusion computation, we must transform the view-cell based visibility algorithm to a view-point based rendering algorithm. to quickly compute the visibility. We, hence, use occluder shrinkage to achieve this.

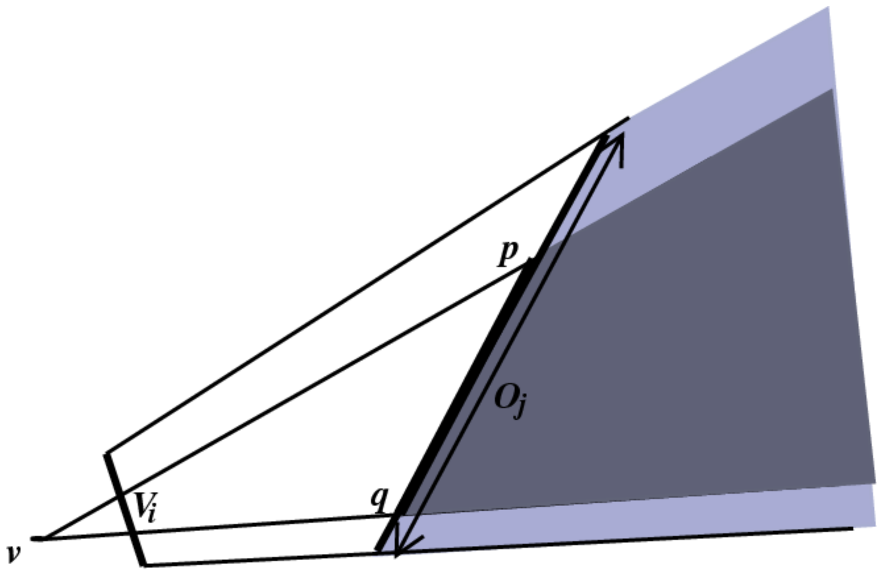


Figure 6: View-cell visibility computed from a point

Consider the example in Fig. 6, with view-cell V_i and occluder O_j. If we construct the supporting planes [CT97], that remain tangent to both V_i and O_j, we obtain a shadow area (light shaded area) contained in the umbra. Now consider point v_i contained within the supporting planes. If we draw planes passing through v_i 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_i, 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 [DDTP00]'s and we do not need to expand the occludees. Neither do we need to generate a large number of samples as in [WWS00].

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 the intersection of the frusta by maximizing the sum of volumes of all shrunk frusta.

5.2.1 Optimization of the projection point, v

Having computed the shadow frustum for each of the connected components, we proceed to find the optimal rendering point from which these shadow frustum will be rendered. We formulate it as a convex quadratic optimization equation. The *rendering point* has the following properties:

- For each of the connected components, the *rendering point* must lie inside its shadow frustum.
- The shadow frustum, when rendering from this *rendering point* must maximize the amount of geometry lying completely inside them.

The last condition above is computationally intensive to solve for. Hence we adopt a conservative approach. Assuming a uniform density of geometry in the world space, we formulate our optimization function as maximizing the sum of the volumes inside each of the shadow frustum. In case the density function of the geometry is given (or computed), we can scale up the individual volumes by their density functions to be the representative of the actual geometry lying in them. The details are are follows:

Let us replace each geometry component C_i by C_i' such that C_i' is a planar approximation of C_i. Let the equation of the plane (C_i') be $\hat{N}_i.X = D_{i1}$, where $\hat{N}_i$ is the unit normal facing toward the view-cell, and D_{i1} is some constant. Let $\hat{N}_i.X = D_{i2}$ be the equation of the plane that just lies inside **W**, the bounding box of the world. Also, for each connected component, we can compute its optimal rendering point (O_i), which is simply the nearest point (I_{ij}) to the connected component C_i. We would want our final rendering point to be as close to O_i as possible. Let $\hat{d}_{i1}$, .., $\hat{d}_{in}$ represent the n direction vectors (each making an angle of greater than $\frac{\pi}{2}$ with $\hat{N}_i$. The volume of the shadow frustum equates to $\frac{1}{3}\ \alpha\ H_2^3 - H_1^3$, where H_2 and H_1 are the heights of O_i from the two planes respectively, and α is a function of $\hat{N}_i, \hat{d}_{i1}, .., \hat{d}_{in}$.

In terms of O_i, the volume is represented as $H_1 = \hat{N}_i.O_i - D_{i1}$ and $H_2 = \hat{N}_i.O_i - D_{i2}$.

Hence Volume = $\frac{1}{3}\alpha(D_{i1} - D_{i2})(3(\hat{N}_i.O_i)^2 - (\hat{N}_i.O_i)(3D_{i1} + 3D_{i2}) + D_{i1}^2 + D_{i2}^2 + D_{i1}D_{i2})$

Let the optimal *rendering point* be represented as **OR**. Hence the volume expression becomes
$\frac{1}{3}\alpha(D_{i1} - D_{i2})(3(\hat{N}_i.\mathbf{OR})^2 - (\hat{N}_i.\mathbf{OR})(3D_{i1} + 3D_{i2}) + D_{i1}^2 + D_{i2}^2 + D_{i1}D_{i2})$ Hence, the net optimization equation becomes:

Maximize $(\Sigma_i \frac{1}{3}\alpha(D_{i1}-D_{i2})(3(\hat{N}_i.\mathbf{OR})^2-(\hat{N}_i.\mathbf{OR})(3D_{i1}+3D_{i2})+D_{i1}^2+D_{i2}^2+D_{i1}D_{i2}))$
s.t. $\Sigma_i(\Sigma_j(A_{ij} \leq B_{ij}))$

However, the above equation is not guaranteed to give an optimal solution in polynomial time. We adopt the following approach to formulate a convex equation which is guaranteed to have an optimal global solution. Instead of maximizing the volume of the newly obtained frustum, we minimize the distance of the new rendering point **OR** from each of the O_i, and scale the distances by the height of the shadow frustum. The net equation becomes:

Minimize $(\Sigma_i\alpha(D_{i1}-D_{i2})((\hat{N}_i.\mathbf{OR})^2+(\hat{N}_i.O_i)^2-2(\hat{N}_i.\mathbf{OR})(\hat{N}_i.O_i)\ )$
s.t. $\Sigma_i(\Sigma_j(A_{ij} \leq B_{ij}))$

The time complexity of solving the above optimization equation is proportional to the number of inequalities, which in our case are the chosen occludes. The following modification reduces the running time to constant time. The basic idea is as follows:

The number of inequalities in the above optimization equation equals O($N * n_{max}$), where n_{max} is the maximum number of edges in any of the connected component. This clearly blows up the running time complexity. We reduce these number of inequalities to O($\Sigma_i degree_i$, $i\varepsilon[1..M]$). Basically, for each edge emanating from a vertex, we compute the intersection of each shadow frustum plane with that edge, and maintain the closest point on that edge. After iterating through all the half planes, we compute for each plane of the view-cell, the closest plane (or say t-2 planes, if there are t points on that plane). Thus the running time complexity reduces to O(t^2), which is constant for a given cell.

Using the above optimization decreased the Visible Set of most of our view-cells by 5% - 10%. The time taken for setting up the optimization equation was around 0.02 seconds.

5.2.2 Occlusion Query

An occluder may be a polygon or a collection of connected polygons. We first decompose the model into connected components and generate the shadow frustum for each component. Our current implementation has only been tested with convex occluders but concave occluders may also be used as long as the shadow frustum can be conservatively computed [BNSVR01]. Our visibility computation algorithm first selects all components larger than the view-cell size as potential occluders. Smaller occluders have a converging shadow volume and hence shrink to NULL. We partition occluders into clusters based on their direction from v to simplify rendering. To compute visibility with respect to each cluster, we first find the projection point v. 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 are 'visible list'. This visible list progressively refined by rendering with the successive occluder cluster.

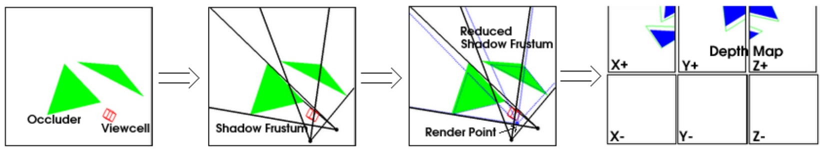


Figure 7: 3D Visibility Pipeline

Note that 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 1 and hence are not updated with the depth value of the rendered occluders.

5.3 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 [KMGL99]. 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.

Before we proceed to our algorithm, we describe an algorithm to compute a bounding cone (represented by an axis and half-angle) for a collection of unit vectors. It will be used as a subroutine frequently in our culling algorithm.

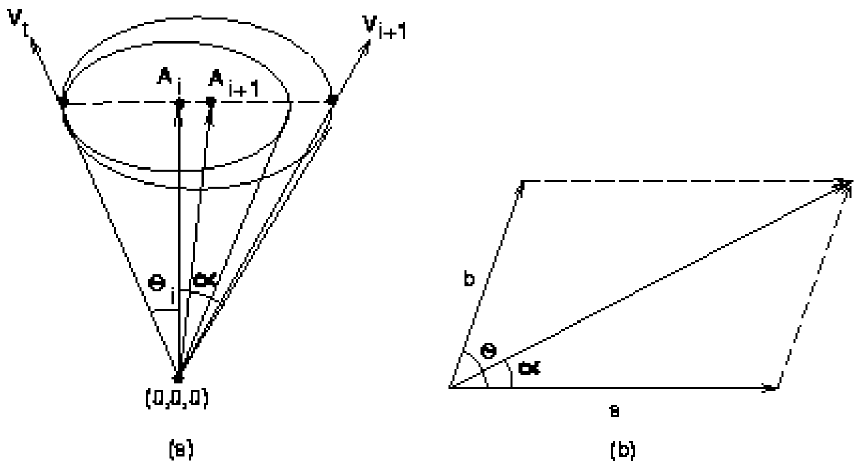


Figure 8: (a) Incremental computation of shell angle and axis vector (b) Illustration of vector addition by parallelogram rule

Computing bounding cones for a set of unit vectors: The bounding cone computation is very similar to the one presented by Sederberg et al.[SM88]. However, the result published there has a minor error in it. We have taken the liberty of correcting it here. Figure 8(a) is also taken from [SM88]. The bounding cone obtained from this algorithm is not an optimal one, but it is near-optimal and linear in time complexity.

Let us assume that after i iterations of the algorithm, our unit axis vector is $\mathbf{A}_i$ with half angle θ_i. In this discussion, we represent the normalized form of a vector $\mathbf{v}$ by $\hat{\mathbf{v}}$. Given a set of n vectors $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_n$, we initialize $\mathbf{A}_1$ to $\hat{\mathbf{V}}_1$ and θ_1 to 0. Each subsequent vector $\mathbf{V}_{i+1}$ is checked to see if it lies within the cone. The vector lies within the cone if $\cos\theta_i \leq \widehat{\mathbf{V}_{i+1}} \cdot \mathbf{A}_i$. If it lies within the cone, $\mathbf{A}_{i+1} = \mathbf{A}_i$ and $\theta_{i+1} = \theta_i$. Otherwise, a new cone is formed which is the smallest cone containing the old cone and the new vector. Let the angle between the vectors $\widehat{\mathbf{V}_{i+1}}$ and $\mathbf{A}_i$ be α as shown in Figure 8(a). Consider a vector $\mathbf{V}_t$ that lies in the same plane as $\mathbf{A}_i$ and $\widehat{\mathbf{V}_{i+1}}$ and makes an angle θ_i with $\mathbf{A}_i$. Clearly, this vector lies on the previous cone. We want to determine this vector in terms of $\mathbf{A}_i$ and $\widehat{\mathbf{V}_{i+1}}$. From the Figure 8(a), it is clear that $\mathbf{V}_t$ can be expressed as some linear combination of $\mathbf{A}_i$ and $-\widehat{\mathbf{V}_{i+1}}$ (*i.e.*, $\mathbf{V}_t = q\mathbf{A}_i - \widehat{\mathbf{V}_{i+1}}$). To determine q, we use the parallelogram law for the sum of two vectors. It is shown in Figure 8(b).

Consider two vectors $\mathbf{a}$ and $\mathbf{b}$ at an angle θ. Let the vector $\mathbf{a} + b$ form an angle α with $\mathbf{a}$. From the triangle law, we know that

$$\frac{\| \mathbf{a} \|}{\| \mathbf{b} \|} = \frac{\sin(\theta - \alpha)}{\sin\alpha}$$

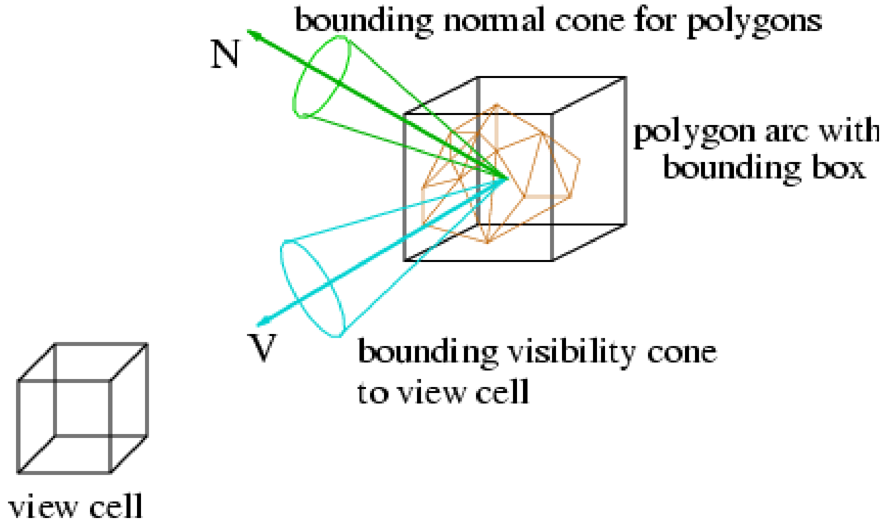

bounding normal cone for polygons
N
polygon arc with bounding box
V
bounding visibility cone to view cell
view cell

Figure 9: Cell-based hierarchical back-face culling

In our case, q, which is the ratio of the lengths of $\mathbf{A}_i$ and $-\hat{\mathbf{V}_{i+1}}$, is given by

$$q = \frac{\sin(\theta_i + \alpha)}{\sin\theta_i}$$

Given $\mathbf{V}_t$, we compute the new axis of the cone

$$\mathbf{A}_{i+1} = \frac{\hat{\mathbf{V}}_t + \hat{\mathbf{V}_{i+1}}}{\| \hat{\mathbf{V}}_t + \hat{\mathbf{V}_{i+1}} \|}$$

and the angle is given by $\cos\theta_{i+1} = \mathbf{A}_{i+1} \cdot \hat{\mathbf{V}_{i+1}}$.

$\mathbf{A}_n$ and θ_n give the axis and half angle of the cone bounding all the original vectors.

We are now ready to describe our back-face culling algorithm. We start with a collection of polygons, its axis-aligned bounding box, B, and a view cell (see Figure 9). Since we have the normals for each of the polygons, we use the above routine to compute a bounding cone, $(\hat{\mathbf{N}}, \theta)$, for all the normal vectors. This bounding normal cone can have its apex anywhere inside B (by construction). 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. It is easy to see that these rays can be bounded by another cone whose apex is at v and bound the vectors from v to the vertices of the view cell. We call this cone the *visibility cone*, $(\hat{\mathbf{V}}_v, \alpha_v)$. Figure 9 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 $\Gamma_v = \cos^{-1}\hat{\mathbf{N}} \cdot \hat{\mathbf{V}}_v + \alpha_v + \theta$ and $\gamma_v = \cos^{-1}\hat{\mathbf{N}} \cdot \hat{\mathbf{V}}_v - \alpha_v - \theta$, the maximum and minimum angle between two vectors, one chosen from 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 polygons classified as *front-facing*, *back-facing*, or *neither*. We remove the back-facing set alone and use the rest as potential front-facing polygons.

6 Implementation and Results

We have implemented our vLOD system and tested it on a detailed 3D powerplant model (13 million triangles, Fig. 1) and a notional city model (Figs. 13-15 consisting of 52.4 million triangles). We have tested our implementation extensively on a Silicon Graphics Onyx2 Infinite Reality workstation and a 2.8GHz Intel PC with an NVIDIA Geforce4 graphics card. The city model has high occlusion from many cells, but due to a large number of small polygons and open areas in the powerplant model, this is a particularly tough case for occlusion culling.

In addition to the visibility computation algorithm described in Section 5, we also implemented the dual ray-space algorithm of [KCC01]. While the rendering time performance was similar, the preprocessing time with the other scheme was much higher and is not reported here. This visibility computation assumes 2.5D occluders. Our implementation preselects occluders based on polygon sizes.

The images generated are the same as if the visibility and simplification were computed at rendering time. Our method does not discard any visible geometry and uses an appropriate level of detail for visible geometry. We have performed experiments to check the coherence. In typical walkthroughs, one only needs to cross a boundary in less than 10% of the frames on average. Most frames require 0 or 1 boundary crossing. Hence, 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's from the disk. Figure 10 shows a frequency diagram of the ratio between Δ and vLOD in a typical walkthrough of the city model. Less than 10% of the time is $\Delta(C)$ more than half the size of vLOD(C) for any given cell C.

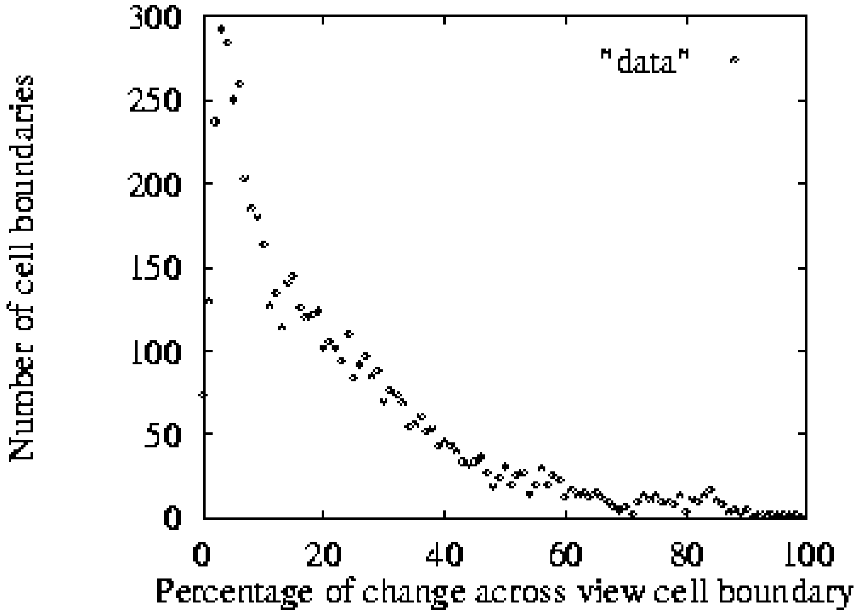


Figure 10: High Coherence: Frequency diagram of ratio of Δ and vLOD

The powerplant model is well known for its complexity. Our city model currently stores no complex texture information but is sufficient to demonstrate the efficacy of our method. The model contains several detail areas and has a highly dynamic range of detail. For example, the cars on the street (see figure 15) each contain over a 100,000 triangles. Several groups of cars are sparsely distributed on the roads of the model. Buildings also contain internal geometry. There is high concentration of highly detailed geometry in the gallery: one of the city buildings. We have only included the powerplant in the video due to its common usage. As can be seen, we are able to maintain interactive frame rate most of the time.

In the precomputation stage, we generated about five million cells for the UNC Powerplant model. It takes four nights on a 16 machines Linux cluster. The time to compute vlod is 0.4 to 2.2 seconds per viewcell. The time to shrink occluder is 0.16 seconds, while rendering and performing occlusion query takes about 0.2 to 2 seconds. Our linux cluster can render approximately 3-4 million triangles per second using vertex arrays, and 50-70K occlusion queries per second.

In our hierarchical subdivision scheme, we also employ two stopping criteria which are the number of visible triangles and the level of the octree. About 30% of the viewcell that reach the lowest level octree require further subdivision. Their average vLOD is about 1.5 times our triangles threshold.

We also perform an informal comparison of our cell visibility result versus estimating vLOD by taking 125 sample points inside a viewcell. On average, the overestimates of our cell visibility algorithm is about two to ten times of the sampling's result (which is an underestimate of vLOD for the viewcell).

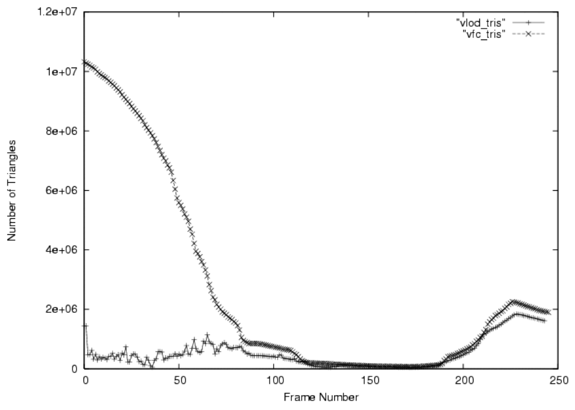


Figure 11: VLOD's triangle count versus view frustum culling

For a particular path through the UNC Powerplant (path 2 in the accompanying video), we visited 180 cells, and had to cross 228 cells. For each of the boundary transition, we fetch around 2 KBytes of data. For the same path, we plot the number of triangles render by our system as compared to the number of triangles render by only view frustum culling. The number of triangles rendered by our system is much less as compared to the number of triangles intersecting the view frustum (Figure 11). We achieve frame rates of around 15-40 frames per second with a screen space error of less than a pixel leading to almost no visual difference.

7 Conclusion

We have presented a new way to combine rendering acceleration techniques 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. Our implementation, even though lacking in many of the optimizations, is able to achieve interactive frame rates on models with around 50 million polygons without using pixel based approximations. This is significantly faster than has been achieved before. Our initial experimentation exhibits a high degree of scalability and suggests

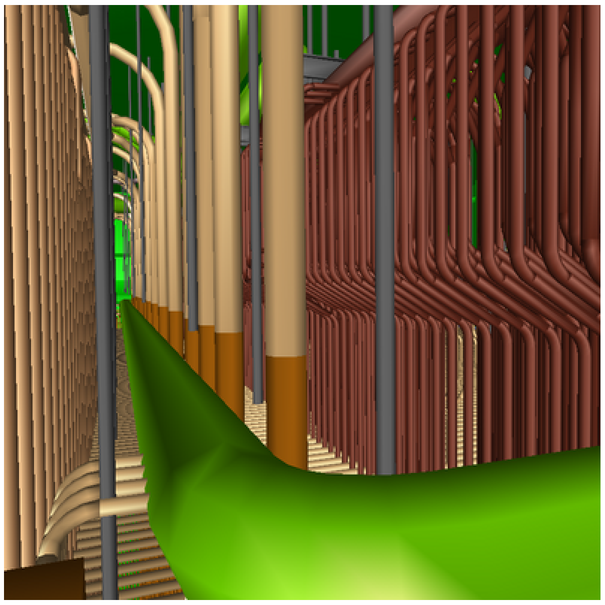


Figure 12: Powerplant Model

that we will be able to display much larger models are interactive rates. We are in the process of acquiring realistic models with over a hundred million polygons. With more accurate visibility pre-computation 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 city 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. This size can be reduced significantly by performing more aggressive occlusion culling and as well as 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 vLOD. 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 improve the performance of this system. We currently allow only limited update of detail at the rendering time. In future, we want to expand that capability. 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.

References

[ACW$^+$99] 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. In *Proc. Symposium on Interactive 3D Graphics*, pages 101–106, Atlanta, GA, 1999.

[AKNW93] F. Alizadeh, R. M. Karp, L. A. Newberg, and D. K. Weisser. Physical mapping of chromosomes: A combinatorial problem in molecular biology. In *ACM/SIAM Symposium on Discrete Algorithms*, pages 371–381, 1993.

[ASCE02] N. Aspert, D. Santa-Cruz, and T. Ebrahimi. Mesh: Measuring errors between surfaces using the hausdorff distance. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, volume I, pages 705 – 708, 2002. `http://mesh.epfl.ch`.

[ASVNB00] Carlos Andújar, Carlos Saona-Vázquez, Isabel Navazo, and Pere Brunet. Integrating occlusion culling and levels of detail through hardly-visible sets. *Computer Graphics Forum*, 19(3):C187–C194, August 2000.

[BDS$^+$01] R. Bukowski, L. Downs, M. Simmons, C. Squin, and S. Teller. Citywalk: A second generation walkthrough system. In *7th International Conference on Virtual Systems and Multimedia (VSMM)*, October 2001.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[BL76] K.S. Booth and G.S. Leuker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[BNSVR01] P. Brunet, I. Navazo, C. Saona-Vázquez, and J. Rossignac. Hoops: 3d curves as conservative occluders for cell-visibility. *Computer Graphics Forum*, 20(10), 2001.

[CCOZ98] Yiorgos Chrysanthou, Daniel Cohen-Or, and Eyal Zadicario. Viewspace partitioning of densely occluded scenes. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry (SCG'98)*, pages 413–414, New York, June 1998. Association for Computing Machinery.

[CDF$^+$00] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. In *ICDE*, pages 489–499, 2000.

[CK01] J. Chhugani and S. Kumar. View-dependent adaptive tessellation of parametric models. In *Proc. Symposium on Interactive 3D Graphics*, pages 59–62, Chapel Hill, NC, 2001.

[CKS02] W. Correa, J. Klosowski, and C. Silva. Out-of- core rendering of large models. Technical Report TR-653-02, Princeton University, 2002.

[CMO97] J. Cohen, D. Manocha, and M. Olano. Simplifying polygonal models using successive mappings. In *IEEE Visualization*, pages 395–402, 1997.

[COM98] J. Cohen, M. Olano, and D. Manocha. Appearance preserving simplification. In *Proc. ACM SIGGRAPH*, 1998.

[Cro77] F. Crow. Shadow algorithms for computer graphics. In *Proc. ACM SIGGRAPH*, pages 242–248, 1977.

[CRS98] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[CT97] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. Symposium on Interactive 3D Graphics*, pages 83–90, 1997.

[DDP96] Frédo Durand, George Drettakis, and Claude Puech. The 3D visibility complex: A new approach to the problems of accurate visibility. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 245–256, New York City, NY, June 1996. Eurographics, Springer Wien. ISBN 3-211-82883-4.

[DDP97] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 89–100. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[DDTP00] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. In *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 239–248. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[ESSS01] J. El-Sana, N. Sokolovsky, and C. Silva. Integrating occlusion culling with view-dependent rendering. In *Proc. IEEE Visualization*, pages 371–378, August 2001.

[FMP97] Leila De Floriani, Paola Magillo, and Enrico Puppo. Building and traversing a surface at variable resolution. In *IEEE Visualization ’97*, October 1997.

[FMP98] Leila De Floriani, Paola Magillo, and Enrico Puppo. Efficient implementation of multi-triangulations. In *Proceedings IEEE Visualization’98*, pages 43–50. IEEE, 1998.

[GCS91] Z. Gigus, J. Canny, and R. Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(6):542–551, 1991.

[GH97] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proc. ACM SIGGRAPH*, pages 209–216, 1997.

[GKM93] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proc. ACM SIGGRAPH*, pages 231–238, 1993.

[GSF99] C. Gotsman, O. Sudarsky, and J. Fayman. Optimized occlusion culling using five-dimensional subdivision. *Computer & graphics*, 23(5):645–654, 1999.

[GSYM02] N. Govindaraju, A. Sud, S. Yoon, and D. Manocha. Parallel occlusion culling for interactive walkthroughs using multiple gpus. Technical Report Technical Report TR02-027, University of North Carolina at Chapel Hill, 2002.

[Gueziec96] A. Guéziec. Surface simplification inside a tolerance volume. Technical report, Yorktown Heights, NY 10598, March 1996. IBM Research Report RC 20440.

[HMC$^+$97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proc. Symposium on Computational Geometry*, 1997.

[Hop96] H. Hoppe. Progressive meshes. In *Proc. ACM SIGGRAPH*, pages 99–108, 1996.

[Hop97] H. Hoppe. View dependent refinement of prograssive meshes. In *Proc. ACM SIGGRAPH*, pages 189–198, 1997.

[HSLM02] K. Hillesland, B. Salomon, A. Lastra, and D. Manocha. Fast and simple occlusion culling based on hardware depth queries. Technical Report Technical Report TR02-039, University of North Carolina at Chapel Hill, 2002.

[JC01] D. Johnson and E. Cohen. Spatialized normal cone hierarchies. In *Proc. Symposium on Interactive 3D Graphics*, pages 129–134, Durham, NC, 2001.

[JM97] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, Ltd., 1997.

[KCC01] Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Hardware-Accelerated from-Region visibility using a dual ray space. In *12th Eurographics Workshop on Rendering*, pages 205–216, 2001.

[KMGL99] S. Kumar, D. Manocha, W. Garrett, and M. Lin. Hierarchical back-face culling. *Computers and Graphics*, 23(5):681–692, 1999.

[LT99] F. Law and T. Tan. Preprocessing occlusion for real-time selective refinement. In *Proc. Symposium on Interactive 3D Graphics*, pages 47–54, 1999.

[ME99] Dinesh Manocha and Carl Erikson. GAPS: General and automatic polygonal simplification. In Stephen N. Spencer, editor, *Proceedings of the Conference on the 1999 Symposium on interactive 3D Graphics*, pages 79–88, New York, April 1999. ACM Press.

[PS99] M.V. Panne and A.J. Stewart. Efficient compression techniques for precomputed visibility. In *12th Eurographics Workshop on Rendering*, Rendering Techniques, pages 305–316, 1999.

[RB93] J.R. Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In *Geometric Modeling in Computer Graphics*, pages 455–465, Genova, Italy, 1993. Spreinger Verlag (eds. B. Falcidieno and T. Kunii).

[SDDS00] G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion. Conservative volumetric visibility with occluder fusion. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 229–238. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[Sim01] M. Simmons. *Tapestry: An Efficient Mesh-based Display Representation for Interactive Rendering*. PhD thesis, UC Berkeley, May, 2001. Technical Report UCB//CSD-01-1153.

[SM88] T.W. Sederberg and R.J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric Design*, 5:161–171, 1988.

[Tel92] S. Teller. Computing the antipenumbra of an area light source. In *Siggraph 1992, Computer Graphics Proceedings*, Annual Conference Series, pages 139–148. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 1992.

[TS91] S. Teller and C. Séquin. Visibility preprocessing for interactive walkthroughs. *ACM Computer Graphics*, 25(4):61–69, 1991. (SIGGRAPH Proceedings).

[VM02] G. Varadhan and D. Manocha. Out-of-core rendering of massive geometric environments. In *Proc. of IEEE Visualization*, 2002.

[WBP98] Y. Wang, H. Bao, and Q. Peng. Accelerated walkthrough of virtual environments based on visibility preprocessing and simplification. *Computer Graphics Forum*, 17(3):C187–C194, 1998.

[WFP$^+$01] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *Siggraph 2001, Computer Graphics Proceedings*, Annual Conference Series, pages 361–370. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2001.

[WS99] Peter Wonka and Dieter Schmalstieg. Occluder shadows for fast walkthroughs of urban environments. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 51–60. The Eurographics Association and Blackwell Publishers, 1999.

[WVBIM02] N. Govindaraju W. V. Baxter III, A. Sud and D. Manocha. Gigawalk: Interactive walkthrough of complex environments. Technical Report Technical Report TR02-013, University of North Carolina at Chapel Hill, 2002. Also in Rendering Techniques '02.

[WWS00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *11th Eurographics Workshop on Rendering*, Brno, Czech Republic, June 2000.

[WWS01] Peter Wonka, Michael Wimmer, and François X. Sillion. Instant visibility. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3) of *Computer Graphics Forum*, pages 411–421. Blackwell Publishing, 2001.

[XV96] J. Xia and A. Varshney. A dynamic view-dependent simplification for polygonal models. In *Proc. IEEE Visualization*, pages 327–334, San Fransisco, CA, 1996.

[ZH97] H. Zhang and K. Hoff. Fast backface culling using normal mask. In *Proc. Symposium on Interactive 3D Graphics*, pages 51–58, Providence, RI, 1997.

[ZMHH97] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. In *Proc. ACM SIGGRAPH*, pages 77–88, 1997.

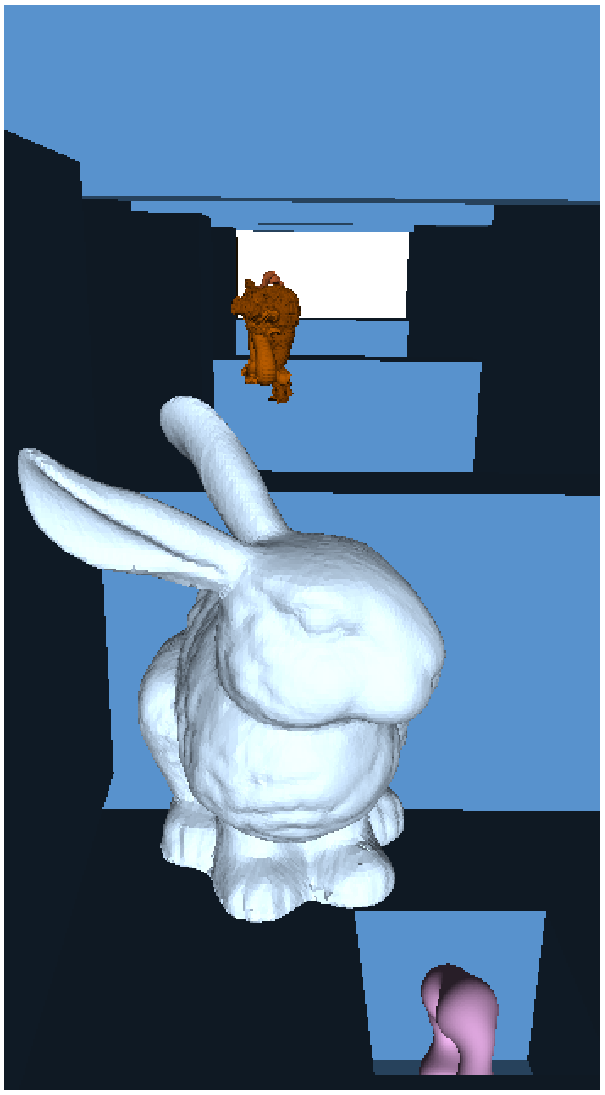


Figure 13: View inside the gallery I

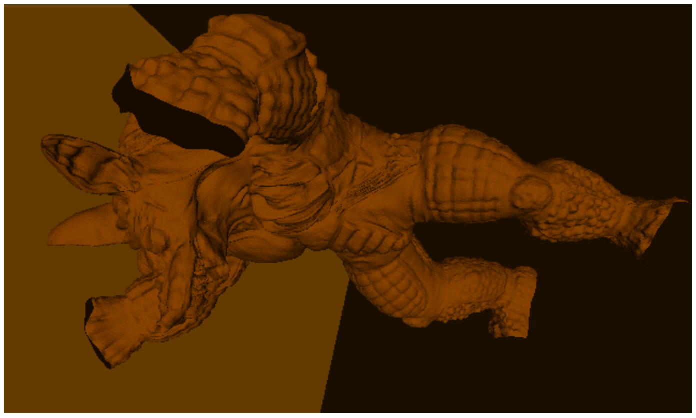


Figure 14: View inside the gallery II

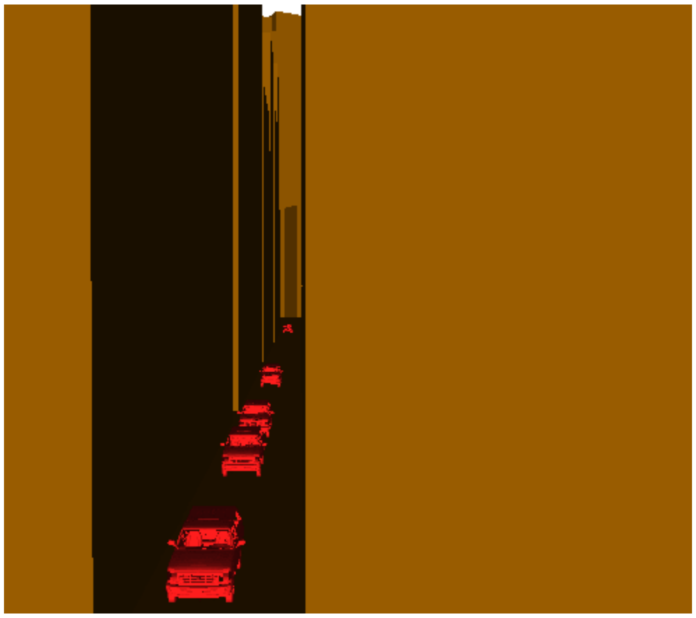


Figure 15: View of cars on the street