Effective Compression Techniques for Precomputed Visibility

Michiel van de Panne
A. James Stewart
Department of Computer Science, University of Toronto
{van,jstewart}@dgp.utoronto.ca

Abstract. In rendering large models, it is important to identify the small subset of primitives that is visible from a given viewpoint. One approach is to partition the viewpoint space into viewpoint cells, and then precompute a visibility table which explicitly records for each viewpoint cell whether or not each primitive is potentially visible. We propose two algorithms for compressing such visibility tables in order to produce compact and natural descriptions of potentially–visible sets. Alternatively, the algorithms can be thought of as techniques for clustering cells and clustering primitives according to visibility criteria. The algorithms are tested on three types of scenes which have very different structures: a terrain model, a building model, and a world consisting of curved tunnels. The results show that the natural structure of each type of scene can automatically be exploited to achieve a compact representation of potentially visible sets.

1 Introduction

The visibility problem is to determine which scene elements are visible from a particular viewpoint. Algorithms that solve this problem are very expensive in time, in memory, or in complexity. For these reasons, real–time applications will often precompute visibility information, store it, and later use it to accelerate rendering.

Storage of the precomputed visibility information can require a very large amount of memory. This paper describes two effective techniques to compress precomputed visibility information. These techniques are very general and may be used in applications as varied as architectural walkthroughs, terrain flyovers, and tunnel roaming.

Techniques that precompute visibility typically divide space into regions and determine what parts of scene are visible from each region. This is a common strategy: Teller and Sequin [17] divide a building into rooms and, for each room, compute the other rooms visible from it. Yagel and Ray [19] subdivide space into a regular grid of cells and use a different method to compute cell–to–cell visibility. Games in maze–like environments often have room–to–room visibility explicitly stored in a table.

These techniques can be thought of as using a very coarse form of clustering to reduce memory requirements: By storing cell–to–cell visibility, rather than cell–to–polygon visibility, groups of polygons in the same cell are clustered and do not need to be exhaustively enumerated.

This coarse form of clustering has several drawbacks: polygon clusters are restricted to correspond one–to–one to viewpoint regions; viewpoint regions themselves are not clustered at all; a polygon cluster must be entirely rendered if even a tiny fraction of it is visible; and it is unclear how to create optimal clusters in less well-structured environments, such as terrains.

In this paper, the space of viewpoints is divided into small regions and a precomputation step determines which *polygons* are visible from each region. A boolean **visibility table** encodes this information: entry (i, j) of the table is TRUE if polygon j is potentially visible from some point in region i. Given the fine subdivision of space and the possibly large number of polygons, the visibility table is potentially huge.

This paper's principal contribution consists of two methods to compress the visibility table:

- The first is a **lossy compression method** which merges viewpoint regions and merges polygons. This method may conservatively deem a polygon to be visible when in fact it is not. Like all conservative visibility algorithms, this does not pose a problem as long as hidden surface elimination (e.g. Z–buffering) is performed during rendering.
- The second is a **lossless compression method** which contructs a graph of viewpoint and polygon clusters. Visible polygons can be enumerated by performing a very simple traversal of this graph. This lossless method never mistakenly deems a polygon to be visible when it is not.

These compression methods have several desirable features:

- A combination of the two compression techniques yields better compression than either alone.
- The level of compression may be chosen to optimize memory, occlusion information, or some ratio of the two.
- These techniques permit very efficient "random access" decompression: For any particular viewpoint region, all visible polygons can be quickly enumerated.
- The polygon and viewpoint clusters are automatically adapted in a natural way to the environment, making this a very general method. For example, in our experiments (presented in Section 6) we discovered:
  - in terrains, polygons are clustered in separate valleys and on peaks;
  - in tunnels, viewpoints are clustered in contiguous tunnel sections; and
  - in buildings, polygons are clustered around "open corridors" from which all of the polygons of the cluster are visible.

The beauty of using the visibility table is that viewpoint clusters and polygon clusters may be treated identically: one consists of a cluster of rows, while the other consists of a cluster of columns. This observation yields very simple algorithms which do not need to know anything about the underlying structure of the viewpoint regions or the scene polygons.

2 Related Work

In work of similar spirit to ours, Yagel and Ray [19] precompute visibility information for a two–dimensional scene using a regular subdivision of space. Their principal contribution is an elegant algorithm to compute cell–to–cell visibility, but they also suggest clustering cells of similar visibility using criteria like those of our lossy compression algorithm. Wang et al.[18] combine precomputed potentially-visible sets with detail simplification in regions where the sets become very large.

Most methods that precompute visibility divide the viewpoint space into cells and compute cell–to–cell visibility. This has the implicit effect of clustering polygons in each cell, which reduces the memory requirement at the cost of not taking advantage of detailed visibility information. Teller and Sequin [17] divide a building into rooms and compute room–to–room visibility. Coorg and Teller[6] exploit the presence of large occluders to perform occlusion culling for a viewpoint. Cohen-Or et al.[2] exploit large convex occluders to compute cell-to-object visibility. Plantinga[13] uses a small set of effective occluders and computes visual events among the occluders in order to partition the viewpoint space into 2D cells.

Coorg and Teller [5], Gigus and Malik [7], and Cohen-Or and Zadicari[4] all exploit features of aspect graphs to produce incremental updates of visibility. Yagel and Ray [19] also suggest recording only changes in visibility in order to compress their cell–to–cell visibility information.

Another class of visibility methods computes visibility during the rendering process. Some examples include the hierarchical Z–buffer of Meagher [12] and of Greene, Kass, and Miller [9], the hierarchical coverage masks of Greene [8], and the hierarchical occlusion maps of Zhang *et al* [20]. An advantage of these techniques is that they can cope with dynamic scenes. However, these techniques work best when a set of large occluders can rapidly be identified for the current viewpoint, which is not always possible. These techniques can potentially be used in conjunction with a compressed visibility table, using a table to achieve the same result as a large occluder.

There has also been a substantial amount of work in clustering for global illumination. Hierarchical radiosity [10], for example, imposes a hierarchical structure on the scene surfaces and computes energy transfer between different nodes in this hierarchy. An alternative “hierarchy of uniform grids” is described by Cazals, Drettakis, and Puech [1]. However, the principal expense in global illumination lies in determining whether one surface sees another, and clustering usually occurs *before* visibility is computed, which is opposite to what we do when compressing the visibility table.

3 The Visibility Table

Visibility is encoded in a boolean table, in which each row initially corresponds to one viewpoint cell and each column initially correponds to one polygon. The table ideally encodes the *partial visibility*: the entry in row i, column j is TRUE if and only if polygon j is at least partially visible from some point in cell i. However, our lossy compression can allow some occlusions to be lost, in which case the table will encode a conservative visibility set[2], which is a superset of the exact partial-visibility set.

Any division of viewpoint regions may be used; our experiments used a regular voxel subdivision of space. One could just as well use another subdivision, such as an oct–tree, a binary space partition, or a k–d tree. Similarly, any division of the scene may be used; our experiments used single polygons. One could also pre–cluster polygons manually, given some knowledge of the scene, or one could go in the opposite direction, subdividing very large polygons (e.g. imposters) if it is likely that only a part of the polygon will be visible at any time.

Given a visibility table, rendering is simple: Locate the row corresponding to the region containing the viewpoint and render each polygon whose corresponding column contains the value TRUE. In order that the rendering time be proportional to the number of visible polygons, each row of the table can be stored as a linked list of the TRUE entries, or alternatively, the FALSE entries must be run–length encoded. We choose the latter. A row is represented by a sequence of integers, where each integer is the number of FALSE entries before the next TRUE entry. For example, the sequence 5, 3, 0, 0, 1 indicates that TRUE entries occur in columns 5, 9, 10, 11, and 13:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
0	0	0	0	0	1	0	0	0	1	1	1	0	1	0	0	0	0	...

The run-length encoding scheme described above can be used as a simple low-cost method for storing the visibility table in an easy-to-decode fashion, comparable to conventional sparse-matrix storage techniques. The job of our compression algorithms is to do much better than this, and thus our compression algorithms make use of this run-length encoding only as a last step. It is worth remarking that, unlike conventional sparse-matrix storage, it would also be possible to run-length encode the non-zero (TRUE) elements, given that there is only one assignable non-zero value. Thus, the sequence 5, 1, 3, 3, 1, 1, 5, 1 could indicate that there are 5 FALSE entries, followed by one TRUE entry, 3 FALSE entries, etc. In many situations, however, this will be less compact than simply run-length encoding the FALSE entries, given the expected sparsity of TRUE entries in densely occluded environments. As well, our compression techniques will first produce visibility tables which are even sparser (but encode the same information), and then use a run-length encoding step as a final compression step.

Experiments have shown a preponderance of short runs of FALSE entries. In order to encode these runs with little memory, each run is represented with either one byte or three bytes: A run of 0 to 254 FALSE entries is encoded with a single byte containing the run length, while a run of 255 to 65535 FALSE entries is encoded with a single byte of 255 followed by two bytes containing the run length.

Given a set of viewpoint regions and a set of polygons, the initial visibility table may be constructed in any one of many ways. Since the initial table construction is not a contribution of this paper, we use a fairly naive approach that *samples* visibility using an item buffer: From several points within each viewpoint region and from six directions around each such point, the scene is rendered into an item buffer in which each polygon is assigned a unique colour. A quick traversal of the item buffer determines which polygons are visible. One problem with the item buffer method is, of course, that it is not conservative: Polygons can be missed due to the limited resolution of the item buffer and the discrete viewpoint sampling done within the viewpoint region. However, artifacts caused by such missing polygons will, by definition, likely be small. Another problem is that it is time consuming, since the scene must be rendered many times from many different viewpoints. For the experiments described in Section 6, the visibility tables took several hours to compute using software rendering on 166 Mhz PC. Other methods may be used to construct the initial visibility table using general methods[19, 15, 2], or methods for specific environments[16, 3].

4 Lossy Compression Algorithm

The lossy compression algorithm compresses the visibility table by merging rows and by merging columns.

- Two rows are merged if they have a similar set of TRUE entries. The merge deletes the two old rows and inserts a new row that is the logical OR of the original two rows. This corresponds to merging two viewpoint regions that have similar sets of visible polygons. The merge creates a new meta–region from which is visible the union of the polygons visible from the two original regions. The logical OR maintains conservative visibility: If a polygon was visible from one of the original regions, it is visible from the merged region.
- Two columns are merged if they have a similar set of TRUE entries. The merge

deletes the two old columns and inserts a new column that is the logical OR or the original two columns. This corresponds to merging two polygons that have identical visibility status in most viewpoint regions. That is, for most viewpoint regions either both polygons are visible or neither is visible. The merge creates a new meta–polygon which is visible from all regions from which either one of the original polygons was visible. Again, the logical OR maintains conservative visibility.

To determine which 'similar' sets of columns or rows should be merged, the lossy algorithm must determine the benefit and the cost of a potential merge.

The **benefit** of a merge is equal to the reduction in table size. Since the table is run–length encoded, its size is proportional to the number of TRUE entries and the benefit of a merge is equal to the number of TRUE entries that are eliminated.

The **cost** of a merge is equal to the number of occlusions that are "lost." An occlusion is represented by a FALSE entry, which indicates that some polygon is occluded from viewpoints in some region. An occlusion is "lost" whenever the logical OR is applied to two entries of *different* boolean values, TRUE and FALSE: The result of the OR is TRUE, and the original FALSE value, representing an occlusion, is lost. Note that no information is lost when two entries of equal value are ORed.

If a row or column is the result of n prior merges, each FALSE entry in that row or column represents n occlusions. If such a FALSE entry is lost in a merge, the cost is n lost occlusions, rather than one. Thus, in order that costs be correctly calculated, each row and column must record the number merges of which it is the result.

Our **slow greedy algorithm** evaluates all pairs of rows and all pairs of columns, and merges the pair with the largest ratio of benefit–to–cost. This is repeated until some user–determined criterion is satisfied. This algorithm is slow because each iteration takes time quadratic in the number of rows and in the number of columns. For large scenes with more than a few thousand initial viewpoint regions or initial polygons, the slow algorithm is unusable. However, the good feature of the slow algorithm is that it performs progressive compression, thereby allowing control over the compromise between lost occlusions and table size. A user might express their desired compromise in terms of a target table size or a limit on the percentage of lost occlusions. We use the latter in our experiments, stopping the lossy compression when 5% of occlusions have been lost.

An alternative **fast greedy algorithm** is used in practice, as follows: A fixed fraction of the rows are chosen as "seed rows." Each remaining "non–seed row" is tested against each seed row and is merged with the seed row of maximum benefit–to–cost ratio. A similar procedure is then performed with the resulting columns. In practice, we choose the seed rows using a regular sampling pattern. If the list of scene primitives has some structure, as we might expect, then this helps to distribute the seed rows in an equitable fashion around the scene. The fast greedy algorithm produces excellent compression and was used in all the lossy compression experiments reported in Section 6.

5 Lossless Compression Algorithm

An alternative compression algorithm can take advantage of locally-similar but globally-dissimilar visibility relationships. For example, viewpoints located in the hallway of a building all share the visibility of a room at the end of the hallway, but they may not share the visibility of rooms located along the hallway. Similar scenarios also occur in non-architectural scenes, as will be illustrated by the experimental results. In terms of the visibility table, these situations correspond to rows or columns that share a large number of TRUE values, but that are not merged by the lossy algorithm because they also differ in a large number of other entries (i.e. their merge cost is too high). The lossless compression algorithm, identifies these situations and merges *only part* of the row or column.

The **lossless compression algorithm** operates as follows (refer to Figure 1):

1. Find a set, V, of viewpoints regions that have a set, P, of visible polygons in common. Pick the set to maximize the product of the cardinalities: $|V| \times |P|$.
2. Create a single polygon cluster consisting of the polygons of P and allocate for it a *new column* in the visibility table. Since the polygon cluster is visible from each of the viewpoint regions in V, the new column has a TRUE value in each row corresponding to a region in V.
3. Symmetrically, create a single region cluster consisting of the viewpoint regions of V and allocate for it a *new row* in the visibility table. Since the region cluster sees all of the polygons in P, the new row has a TRUE value in each column corresponding to a polygon in P.
4. Set to FALSE all entries in the intersection of the rows and columns of V and P. (These entries are made redundant with the addition of the new row and column.)
5. Repeat with new V and P until no clusters remain above some user–defined size.

There is no cost to this operation, since no occlusions are lost. The benefit is that the visibility table becomes sparser. For a given cluster, the benefit can be computed as the number of TRUE entries which become FALSE, minus the number of TRUE entries created in the new row and new column. If $|V|$ and $|P|$ are the cardinalities of V and P, respectively, then the benefit is $|V| \times |P| - |V| - |P|$. By including the new region and polygon clusters as a new row and new column of the table, these clusters can participate in subsequent merges, as shown in Figures 1(b) and 1(c).

To determine which polygons are visible from a particular viewpoint region, the corresponding row of the visibility table is traversed, just as it is done with the lossy–compressed table. However, if (on that row) a TRUE entry is encountered in some column, j, and if column j corresponds to a polygon cluster rather than to a single polygon, then a recursive traversal of row j is performed. (It is easy to distinguish the cluster columns, since they all appear to the right of the rightmost polygon column.)

It is interesting to note that the same traversal may be performed with the roles of the rows and columns reversed (i.e. traverse each column). In this case, we enumerate all viewpoint regions that see a particular polygon. This symmetry is evident in Figure 1. For example, polygon c is visible from regions 0, 1, and 3.

6 Experimental Results

Our experiments applied both types of compression, as well as their combination, to three types of scenes. These scenes were chosen for their dissimilar structures in order to illustrate the generality of our method and to show that the compression techniques automatically exploit the natural structure of each scene in order to yield compact visibility descriptions. We first motivate the choice of each dataset or scene and then provide a summary of the key properties of each scene.

The terrain dataset is shown in Figure 2 and was procedurally generated using an iterative subdivide-and-displace approximation of a fractal terrain. The existence of occlusion-culling techniques specific to terrains [14] motivated the choice of this example, as well as the large number of simulation and gaming applications that involve

	a	b	c	d	e	f
0	**1**	**1**	**1**	**1**	0	**1**
1	**1**	**1**	**1**	**1**	1	**1**
2	1	1	0	1	0	1
3	0	1	1	1	0	1

$\longrightarrow$ **Cluster** α

	a	b	c	d	e	f	α
0	0	0	0	0	0	0	**1**
1	0	0	0	0	1	0	**1**
2	1	1	0	1	0	1	0
3	0	1	1	1	0	1	0
α	**1**	**1**	**1**	**1**	0	**1**	0

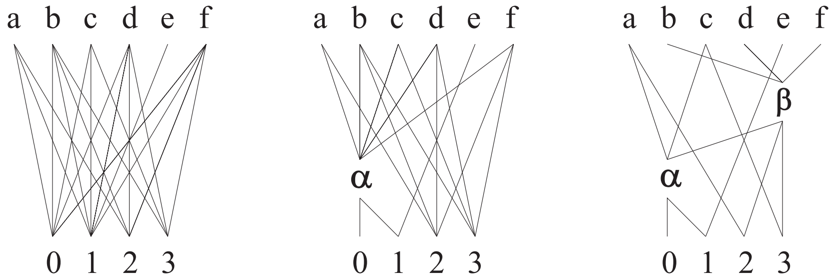
(a)

	a	b	c	d	e	f	α
0	0	0	0	0	0	0	1
1	0	0	0	0	1	0	1
2	1	**1**	0	**1**	0	**1**	0
3	0	**1**	1	**1**	0	**1**	0
α	1	**1**	1	**1**	0	**1**	0

$\longrightarrow$ **Cluster** β

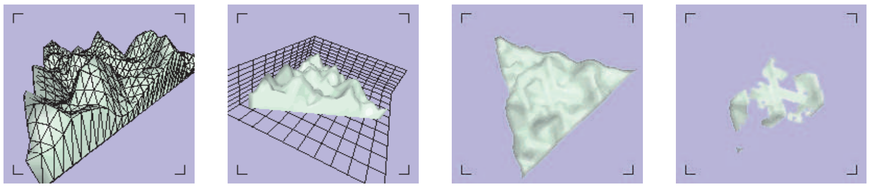
	a	b	c	d	e	f	α	β
0	0	0	0	0	0	0	1	0
1	0	0	0	0	1	0	1	0
2	1	0	0	0	0	0	0	**1**
3	0	0	1	0	0	0	0	**1**
α	1	0	1	0	0	0	0	**1**
β	0	**1**	0	**1**	0	**1**	0	0

(b)



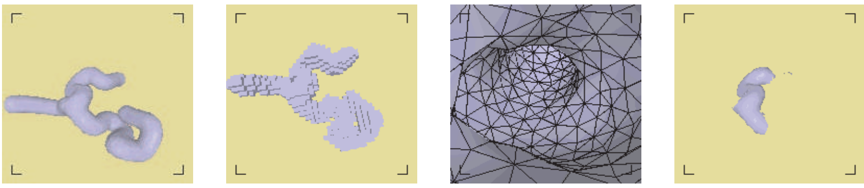
(c)

Fig. 1. An example of the lossless clustering algorithm, where letters denote polygons and digits denote viewpoint regions. **(a)** The visibility table is modified when a cluster, α, is created, consisting of $\{0,1\} \times \{a,b,c,d,f\}$. **(b)** A second cluster, β, is created, consisting of $\{2,3,\alpha\} \times \{b,d,f\}$. Included in β are the viewpoint regions of cluster α, but not the polygons of cluster α. **(c)** An alternative representation of the visibility table, where a line directed *upward* from i to j corresponds to TRUE entry in row i, column j of the visibility table. To enumerate the polygons visible in a region, i, *all upward paths* from i are followed. For clustered tables, these paths traverse intermediate clusters, such as α and β in the rightmost graph. For example, the polygons visible from viewpoint region 3 are *b, c, d*, and *f*. All polygons are visible from region 1.



(a) (b) (c) (d)

Fig. 2. The terrain dataset. (a) primitives (b) voxel grid defining the viewpoint cells (c) plan view of terrain (d) plan view of visible set for a particular viewpoint.



(a) (b) (c) (d)

Fig. 3. The tunnels dataset. (a) exterior view (b) voxelization used for the viewpoint cells (c) interior view with diffuse lighting (d) exterior view of visible set for the viewpoint used in (c).

moving on or over terrains. In our example we deal specifically with voxel–to–primitive visibility, as one might use in a flight simulator. Other applications which restrict the viewpoint to the surface, such as in driving or walking simulations, would do better to use a 2D surface parameterization of the viewpoint space instead of a volumetric parameterization. The viewing space is divided into a $10 \times 10 \times 8$ set of voxels, as shown in Figure 2(b). Viewpoints below the terrain surface are treated as having a completely occluded view of the world.

Our second test scene consists of a set of winding tunnels, as shown in Figure 3. This dataset was motivated by applications such as colonoscopy [11]. The voxelization is constructed procedurally from the boundary representation of the tunnels. The viewpoint cell which contains a given viewpoint can be quickly located by computing a unique voxel ID based on the quantized xyz coordinates of the viewpoint, and then using a hash table indexed by this ID. This avoids allocating voxel storage for the large volume of space outside of the tunnel system, under the assumption that the viewpoint should always remain within the tunnels. Viewpoints that are outside the tunnel walls — even if they are inside cells that straddle the walls — are considered to see nothing, so treatment of straddling cells requires some care.

Lastly, we choose a building floorplan as a dataset, as shown in Figure 4. The floorplan exists on a 10×17 unit grid. All wall segments are broken into primitives, each no longer than 1 unit in length. The viewpoint cells consist of 1×1 unit squares on the grid. These are schematically illustrated in Figure 4(b). The scene is constructed as a 3D model, although the 2D floorplan effectively contains all the information about the structure of the scene. Each wall is double–sided.

The key properties that characterize each of the three example scenes are given in Table 1. Some details of this table bear further explanation. The maximum occlusion in the terrain and tunnel scenes is 100%, as some viewpoint cells are located below the terrain surface or outside the tunnel wall. The dataset size assumes an uncompressed

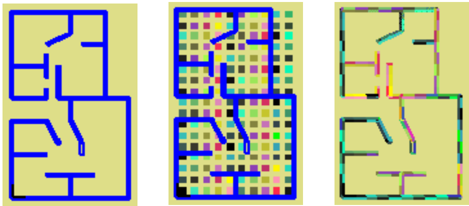


Fig. 4. The rooms dataset. (a) plan view (b) plan view showing voxels (c) plan view showing primitives

scene properties	terrain	tunnels	rooms
# viewpoint cells	800	1275	170
# primitives	1217	4166	242
mean occlusion	84%	82%	90%
min, max occlusion	48%, 100%	70%, 100%	80%, 94%
data set size	15 kB	50 kB	10 kB
uncompressed table size	122 kB	664 kB	5.3 kB
table size, gzip	26 kB	64 kB	1.2 kB
lossy compression			
reduction applied	2p, 2v	10p, 5v	200g
table size (compression)	22.1 kB (5.5×)	23 kB (29×)	0.2 kB (27×)
lossless compression			
# clusters	2000	2000	100
table size (compression)	28.5 kB (4.3×)	95 kB (7.0×)	1.3 kB (4.1×)
lossless compression			
table size (compression)	6.1 kB (20×)	4.4 kB (151×)	0.1 kB (53×)
% of original table size (gzip)	5% (21%)	0.7% (10%)	2% (23%)
% of dataset size (gzip)	41% (173%)	9% (128%)	1% (12%)

Table 1. Experimental Data and Results

representation of an indexed–vertex representation of the geometry. An xyz vertex triple is assumed to be represented in 12 bytes, while an index is assumed to be stored as a 4–byte integer. The raw table size assumes a binary representation of the visibility table. The run–length encoded versions of the raw table are comparable in size to the binary–encoded raw tables for our examples.

We use `gzip` to provide a simple point of comparison for our compression techniques. In practice, the requirement for random access to the rows in order to answer visibility queries means that individual rows should be compressed instead of the entire table. We use `gzip` as applied to the entire table as an upper bound on the performance of a row–based compression algorithm. It should also be noted that a further advantage of our compression schemes is that they produce enumerated lists of visible primitives, unlike binary compression algorithms which only reproduce the original rows of the visibility table. Lastly, we note that gzip is only a 'fair' comparison for the lossless compression algorithm.

6.1 Lossy Compression Results

The compression tests we apply to the test scenes are (1) lossy compression, (2) lossless compression, and (3) lossy compression followed by lossless compression. We first look at how lossy compression performs on the datasets. The results are summarized in Table 1, where the notation `10p, 5v` means that the number of primitives was reduced by a factor of 10 and the number of viewpoint cells was reduced by a factor of 5. In all cases, the lossy clustering procedure was continued until 5% of the occlusions present in the original visibility table were lost. The fast greedy algorithm was applied in the case of the terrain and tunnels datasets because of their size. The slow greedy algorithm could be applied to the room scene (for 200 iterations) because of its small size. In all cases, the lossy compression took less than 20 minutes to complete as computed on a 166 MHz PC.

The results show that the lossy compression scheme can reduce the size of the precomputed visibility table by a factor of 29 for the well–structured tunnel scene. It also does very well on the rooms scene ($27\times$ compression), although not as well for the less–structured terrain scene ($5.5\times$). Figure 5(a) (see color plates) shows the clusters which are formed for the terrain scene, resulting from reducing the number of primitives by a factor of 30, showing a plan view on the left, and an interior view on the right. Clusters typically consist of connected polygons, although this adjacency information is not explicitly present in the raw visibility table. As well, clusters tend to be based on mountain faces and valleys. The clusters produced for the tunnel scene, shown in Figure 5(b) show similar behaviors, creating a patchwork coverage of the tunnel walls. We do not show the viewpoint clusters for the terrain and tunnel scenes, as they are difficult to depict, given their 3D nature.

The viewpoint clusters formed for the room scene, shown in Figure 5(c) are mainly based on individual rooms, as one might intuitively expect. The primitive clusters are similarly organized. The two figures on the right illustrate examples of the lost occlusions that arise during the lossy compression. The viewpoint is indicated as a blue dot, while the blue wall segments indicate the original minimal–size PVS. The red wall segments indicate the additions made to the PVS in the interest of obtaining good compression.

6.2 Lossless Compression Results

The results for lossless compression are shown in Table 1 and Figure 6. Lossless clustering by itself produces compression factors of between 4 and 7. These results are comparable to `gzip`, keeping in mind that `gzip` is not amenable to the random–access and fast enumeration requirements of visibility applications.

The clusters formed by the lossless compression algorithm have a different structure than those of the lossy compression, as shown in Figure 6 (see color plates). Clusters formed early on appear high in the cluster hierarchy and result in the largest storage reductions. We illustrate some of these clusters in Figure 6 for the three test scenes. In Figure 6(a), the same cluster is shown twice; first in a plan view and then in a side view. Although the cluster looks to be an unlikely agglomeration of primitives in the plan view, the side view reveals that it is really a group of primitives facing a particular direction, and hence consistently visible as a group from a particular (large) set of viewpoints. The important clusters for the tunnel scene, shown in Figure 6(b), tend to be coherent cylindrical regions of the tunnels, although the clusters do not provide completely continuous coverage, as evident from the "holes." The appearance of these holes is unintuitive to us. The rooms dataset provides results having an intuitive interpretation, as shown in Figure 6(c). The figure on the left shows a cluster which groups together the primitives in a room. The remaining two clusters illustrate distributed clusters which effectively capture particular 'visibility corridors' within the scene.

6.3 Combining Lossy and Lossless Compression

An interesting characteristic of the lossy and lossless compression techniques is that they appear to operate in orthogonal fashions. The results of following lossy compression with lossless compression are given in Table 1. The occlusions which are 'lost' when using the combination of techniques are the same as those which are lost using the lossy technique alone. The compression ratios achieved for the terrain and tunnels scenes are approximately the product of the compression ratios achievable by each technique alone. For all three datasets, the combined compression performs 4 to 10 times better than `gzip`. More interestingly, the final compressed visibility table, only amounts to a small percentage of the storage space required for the uncompressed dataset geometry.

There are several caveats to be stated about Table 1. The rooms example has 3D geometry but a 2D visibility problem, and therefore the storage cost for the precomputed visibility is small when compared to the storage cost for the dataset geometry. The tunnels dataset is perhaps the most convincing result, storing 95% of the occlusion relationships with a memory cost of 9% of the uncompressed dataset geometry. However, this kind of results is probably restricted to scenes which have well structured visibility coherence. A last caveat is that any implementation that must decode the table must also build associated indexing data structures such as the hash table required for the tunnel scene in order to efficiently locate the viewpoint cell, given a particular viewpoint. These additional data structures could potentially double or triple the storage costs given in Table 1, although we have not yet evaluated their average costs. Nevertheless, the results show that remarkably little storage space is needed to encode precomputed visibility information.

7 Summary and Discussion

Precomputed visibility information can be used to answer the question "What is visible from this viewpoint?" The algorithms presented in this paper address a new problem (to the best of our knowledge): How can precomputed visibility information be efficiently compressed? Experimental results show that the proposed algorithms are effective for three very different types of datasets: a terrain, a series of winding tunnels, and a building interior. The result is a near–optimal method of occlusion culling which has low storage cost and which permits fast, random–access enumeration of visible primitives.

We propose a variety of future work which would extend these techniques to new application domains as well as addressing some caveats in their use.

- Scalability is an important issue. Large scenes likely need to be tackled with a top–down divide–and–conquer approach, in addition to the bottom–up approach proposed thus far.
- The compromise between storage cost and lost occlusions needs to be explored in terms of a trade-off between rendering time and storage cost.
- The viewpoint space could be expanded to include the viewing direction. The technique could also be applied to efficiently store voxel–to–voxel visibility.
- Models of the rendering cost could be incorporated into the clustering criteria.

References

1. Frédéric Cazals, George Drettakis, and Claude Puech. Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. *Computer Graphics Forum*, 14(3):371–382, August 1995. Proceedings of Eurographics '95. ISSN 1067-7055.
2. D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. 1998.
3. D. Cohen-Or and A. Shaked. Visibility and dead-zones in digital terrain maps. *Computer Graphics Forum*, 14(3):C/171–C/180, September 1995.
4. D. Cohen-Or and E. Zadicario. Visibility streaming for network-based walkthroughs. In *Proceedings of Graphics Interface '98*, 1998.
5. Satyan Coorg and Seth Teller. Temporally coherent conservative visibility. In *Proceedings of the Twelfth Annual Symposium On Computational Geometry (ISG '96)*, pages 78–87, New York, May 1996. ACM Press.
6. Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *Proceedings of the 1997 Symposium on 3D Interactive Graphics*, 1997.
7. Z. Gigus and J. Malik. Computing the aspect graph for the line drawings of polyhedral objects. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 12(2), February 1990.
8. N. Greene. Hierarchical polygon tiling with coverage masks. *Computer Graphics*, 30(Annual Conference Series):65–74, 1996.
9. N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, 1993.
10. Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):197–206, July 1991.
11. Lichan Hong, Shigeru Muraki, Arie Kaufman, Dirk Bartz, and Taosong He. Virtual voyage: Interactive navigation in the human colon. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 27–34. ACM SIGGRAPH, Addison Wesley, August 1997.
12. Donald J. Meagher. Efficient synthetic image generation of arbitrary 3–D objects. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, pages 473–478, June 1982.
13. H. Plantinga. Conservative visibility preprocessing for efficient walkthroughs of 3d scenes. In *Proceedings of Graphics Interface '93*, pages 166–173, 1993.
14. A. James Stewart. Fast horizon computation at all points of a terrain with visibility and shading applications. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):82–93, March 1998.
15. Seth Teller and Pat Hanrahan. Global visibility algorithms for illumination computations. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 239–246, 1993.
16. Seth J. Teller. Computing the antipenumbra of an area light source. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 139–148, July 1992.
17. Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 61–69, July 1991.
18. Y. Wang, H. Bao, and Q. Peng. Accelerated walkthroughs of virtual environments based on visibility preprocessing and simplification. *Computer Graphics Forum (Eurographics 98 issue)*, 17(3):187–194, 1998.
19. R. Yagel and W. Ray. Visibility computation for efficient walkthrough of complex environments. *Presence*, 5(1):45–60, 1995.
20. H. Zhang, D. Manocha, T. Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings*.

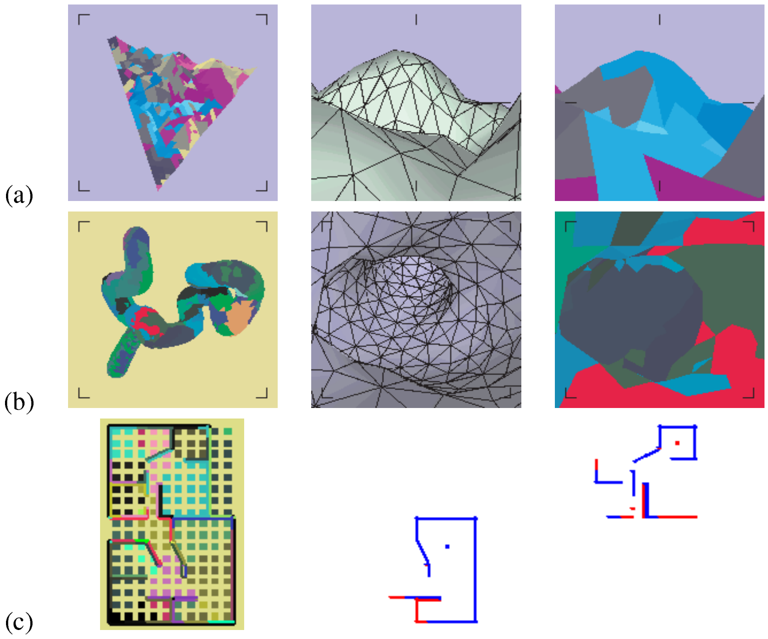


Fig. 5. Clustering resulting from lossy compression. (a) terrain dataset (b) tunnels dataset (c) rooms dataset

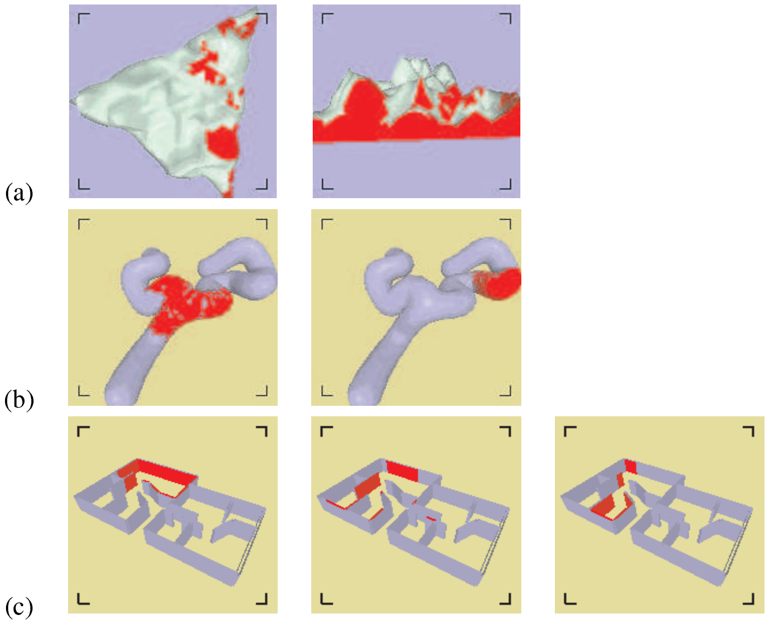


Fig. 6. Clustering resulting from lossless compression. (a) terrain dataset (b) tunnels dataset (c) rooms dataset