View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Rendering on a Budget: A Framework for Time-Critical Rendering

James T. Klosowski* Cláudio T. Silva†

IBM T. J. Watson Research Center

Abstract

We present a technique for optimizing the rendering of high-depth complexity scenes. *Prioritized-Layered Projection* (PLP) does this by rendering an estimation of the visible set for each frame. The novelty in our work lies in the fact that we do not explicitly compute visible sets. Instead, our work is based on computing *on demand* a priority order for the polygons that maximizes the likelihood of rendering visible polygons before occluded ones for any given scene. Given a fixed budget, *e.g.* time or number of triangles, our rendering algorithm makes sure to render geometry respecting the computed priority.

There are two main steps to our technique: (1) an occupancy-based tessellation of space; and (2) a solidity-based traversal algorithm. PLP works by first computing an occupancy-based tessellation of space, which tends to have more cells where there are more geometric primitives. In this spatial tessellation, each cell is assigned a *solidity* value, which is directly proportional to its likelihood of occluding other cells. In its simplest form, a cell's solidity value is directly proportional to the number of polygons contained within it. During our traversal algorithm cells are marked for projection, and the geometric primitives contained within them actually rendered. The traversal algorithm makes use of the cells' solidity, and other view-dependent information to determine the ordering in which to project cells. By carefully tailoring the traversal algorithm to the occupancy-based tessellation, we can achieve very good frame rates with low preprocessing and rendering costs.

In this paper, we describe our technique and its implementation in detail. Also, we provide experimental evidence of its performance. We also briefly discuss extensions of our algorithm.

Key Words and Phrases: Polygon rendering, visibility ordering, occlusion culling.

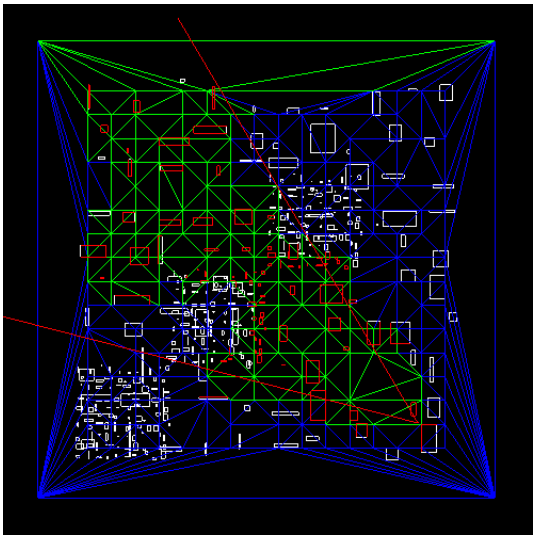


Figure 1: The Prioritized-Layered Projection Algorithm. PLP attempts to prioritize the rendering of geometry along layers of occlusion. The input geometry, line segments in two-dimension, is drawn in white and the spatial tessellation, a Delaunay Triangulation, is drawn in blue. Cells that have been projected by the PLP algorithm are highlighted in green and rendered geometry is drawn in red. The view frustum is also highlighted as red line segments. In this particular example, a budget of 500 line segments was used. The 2D prototype implementation does not enforce star-shaped constraints (or penalties) on the front.

1 Introduction

Recent advances in graphics hardware have not been able to keep up with the increase in scene complexity. In order to support a new set of demanding applications, a multitude of rendering algorithms have been developed to both augment and optimize the use of the hardware. An effective way to speed up rendering is to avoid rendering geometry that cannot be seen from the given viewpoint, such as geometry that is outside the view frustum, faces away from the viewer, or is obscured by some previously rendered geometry. Quite possibly, the hardest part of the visibility-culling problem is to avoid rendering geometry that can not be seen due to its being obscured by previous rendered geometry. In this paper, we propose a new algorithm for solving the visibility culling problem. Our technique is an effective way to cull geometry with a very simple and general algorithm.

Our technique optimizes for rendering by estimating the visible set for a given frame, and only rendering those polygons. It is based on computing *on demand* a priority order for the polygons that maximizes the likelihood of projecting visible polygons before occluded ones for any given scene. It does so in two steps: (1) as a preprocessing step, it computes an occupancy-based tessellation of space, which tends to have more spatial cells where there are more geometric primitives; (2) in real-time, rendering is performed by traversing the cells in an order determined by their intrinsic solidity and some other view-dependent information. As cells are pro-

*Visual Technologies, IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598; jklosow@watson.ibm.com.

†Visual Technologies, IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598; csilva@watson.ibm.com.

jected, their geometry is scheduled for rendering (see Fig. 1). Actual rendering is constrained by a user-defined budget, *e.g.* time or number of triangles.

Some highlights of our technique:

- **Budget-based rendering.** Our algorithm generates a projection ordering for the geometric primitives that mimics a "depth-layered" projection ordering, where primitives directly visible from the viewpoint are projected earlier in the rendering process. The ordering and rendering algorithms strictly adhere to a user-defined budget, making the PLP approach time-critical.
- **Low-complexity preprocessing.** Our algorithm requires inexpensive preprocessing, that basically amounts to computing an Octree and a Delaunay triangulation on a subset of the vertices of the original geometry.
- **No need to choose occluders beforehand.** Contrary to other techniques, we do not require that occluders be found before geometry is rendered.
- **Object-space occluder fusion.** All of the occluders are found automatically during a space traversal that is part of the normal rendering loop without resorting to image-space representation.
- **Simple and fast to implement.** Our technique amounts to a small modification of a well-known rendering loop used in volume rendering of unstructured grids. It only requires negligible overhead on top of view-frustum culling techniques.

Our paper is organized as follows. In Section 2, we give some preliminary definitions, and briefly discuss relevant related work. In Section 3, we propose our novel visibility-culling algorithm. In Section 4, we give some details on our prototype implementation. In Section 5, we provide experimental evidence of the effectiveness of our algorithm. In Section 6, we conclude the paper with some final remarks and future work.

2 Preliminaries and Related Work

The visibility problem is defined in [9] as follows. Let the scene, S, be composed of modeling primitives (*e.g.*, triangles, or spheres) $S = \{P_0, P_1, \ldots, P_n\}$, and a viewing frustum defining an eye position, a view direction, and a field of view. The visibility problem encompasses finding the points or fragments within the scene which are visible, that is, connected to the eyepoint by a line segment that meets the closure of no other primitive. For a scene with $n = O(|S|)$ primitives, the complexity of the set of visible fragments might be as high as $O(n^2)$, but by exploiting the discrete nature of the screen, the Z-buffer algorithm [2] solves the visibility problem in time $O(n)$, since it only touches each primitive once. The Z-buffer algorithm solves the visibility problem by keeping a depth value for each pixel, and only updating the pixels when geometry closer to the eyepoint is rendered. In the case of high-depth complexity scenes, the Z-buffer might overdraw each pixel a considerable number of times. Despite this potential inefficiency, the Z-buffer is a popular algorithm, widely implemented in hardware.

In light of the Z-buffer being widely available, and exact visibility computations being potentially too costly, one idea is to use the Z-buffer as filter, and design algorithms that lower the amount of overdraw by computing an approximation of the *visible set*. In more precise terms, define the visible set $V \subset S$ to be the set of modeling primitives which contribute to at least one pixel of the screen.

In computer graphics, visibility-culling research mainly focussed on algorithms for computing (hopefully tight) estimations of V, then using the Z-buffer to obtain correct images. The simplest example of visibility-culling algorithms are backface and view-frustum culling [11]. Backface-culling algorithms avoid rendering geometry that face away from the viewer, while viewing-frustum culling algorithms avoid rendering geometry that is outside of the viewing frustum. Even though both of these techniques are very effective at culling geometry, more complex techniques can lead to substantial improvements in rendering time. These techniques for tighter estimation of V do not come easily. In fact, most techniques proposed are quite involved and ingenious, and usually require the computation of complex object hierarchies in both 3- and 2-space.

Here again the discrete nature of the screen, and screen-space coverage tests, play a central whole in literally all occlusion-culling algorithms, since it paves the way for the use of screen occupancy to cull 3D geometry that projects into already occupied areas. In general, algorithms exploit this fact by (1) projecting P_i in front-to-back order, and (2) keeping screen coverage information. Several efficiency issues are important for occlusion-culling algorithms:

(a) They must operate under great time and space constraints, since large amounts of geometry must be rendered in 1/30th of a second for real-time display.

(b) It is imperative that primitives that will not be rendered be discarded as early as possible, and (hopefully) not be touched at all. Global operations, such as computing a full front-to-back ordering of P_i, should be avoided.

(b) The more geometry that gets projected, the less likely the Z-buffer gets changed. In order to effectively use this fact, it must be possible to merge the effect of multiple occluders. That is, it must be possible to account for the case that neither P_0 nor P_1 obscure P_2 by itself, but together they do cover P_2. Algorithms that do not exploit *occluder-fusion*, are likely to rely on the presence of large occluders in the scene.

A great amount of work has been done in visibility culling in both computer graphics and computational geometry. For those interested in the computational geometry literature, see [9, 8, 10]. For a survey of computer graphics work, see [20].

We very briefly survey some of the recent work more directly related to our technique. Hierarchical occlusion maps [21] solve the visibility problem by using two hierarchies, an object-space bounding volume hierarchy and another hierarchy of image space occlusion maps. For each frame, a set of objects from a pre-computed database is chosen to be occluders, and used to cull geometry that cannot be seen. A closely related technique is the hierarchical Z-buffer [13]. In [1], an extension of graphics hardware for occlusion-culling queries is proposed.

It is possible to perform object-space visibility culling. One such technique, described in [18], divides space into cells,

which are then preprocessed for potential visibility. This technique works particularly well for architectural models. Additional object-space techniques are described in [6, 7]. These techniques mostly exploit the presence of large occluders, and keep track of spatial extents over time. In [4], a technique that precomputes visibility in densely occluded scenes is proposed. They show it is possible to achieve very high-occlusion rates in dense environments by pre-computing simple ray-shooting checks.

In [12], a constant-frame rendering system is described. This work uses the visibility-culling from [18]. It is related to our approach in the sense that it also uses a (polygon) budget for limiting the overall rendering time. Other notable references include [3], for its level-of-detail management ideas; and [16], where a scalable rendering architecture is proposed.

3 The PLP Algorithm

In this paper we propose the *Prioritized-Layered Projection* algorithm, a simple and effective technique for optimizing the rendering of geometric primitives. The guts of our algorithm consists of a space-traversal algorithm, which prioritizes the projection of the geometric primitives in such a way as to avoid (actually delay) projecting cells that have a small likelihood of being visible. Instead of explicitly overestimating V, our algorithm works on a budget. At each frame, the user can provide a maximum number of primitives to be rendered, a polygon budget (also available is a ρ-budget, to be explained below), and our algorithm, in its single-pass traversal over the data, will deliver what it considers to be the set of primitives which maximizes the image quality, (using a solidity-based metric).

Our projection strategy is completely object-space based, and resembles cell-projection algorithms used in volume rendering unstructured grids. *

In a nutshell, our algorithm is composed of two parts:

Preprocessing. Here, we tessellate the space that contains the original input geometry with convex cells in the way specified in Section 3.1. During this one-time preprocessing, a collection of tetrahedron is generated in such a way as to roughly keep an uniform density of primitives per tetrahedron. Our sampling leads to large tetrahedra in unpopulated areas, and small tetrahedra in areas that contain a lot of geometry.

In another similarity to volume rendering, using the number of modeling primitives assigned to a given cell (*e.g.*, tetrahedron) we define its *solidity* value ρ, which is similar to the opacity used in volume rendering. In fact, we use a different name to avoid confusion since the accumulated solidity value used throughout our priority-driven traversal algorithm can be larger than one. Our traversal algorithm prioritizes cells based on their solidity value.

Generating such a space tessellation is not a very expensive step, *e.g.* taking only two minutes for a scene composed of one million triangles, and for several large dataset can even be performed as part of the data input process. Of course, for truly large datasets, we highly recommend generating this view-independent data structure beforehand, and saving it with the original data.

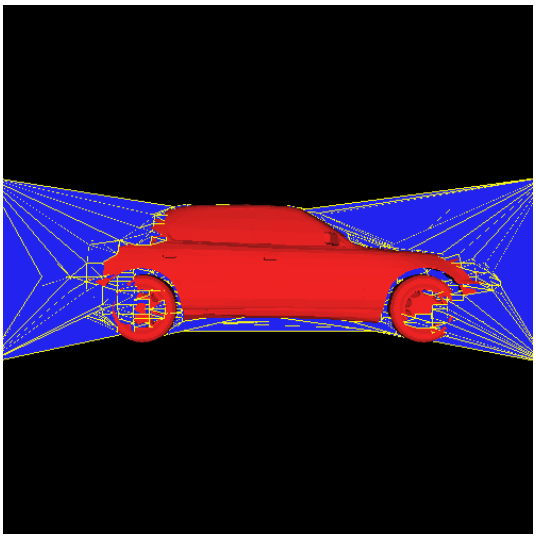


Figure 2: A snapshot of the PLP algorithm. Rendered geometry is shown in red. The cells of the spatial tessellation that are in the front are drawn in blue. The next cell to be projected is the (blue) one with the lowest solidity value.

Rendering Loop. Our rendering algorithm traverses the cells in roughly front-to-back order. Starting from the seed cell, which in general contains the eye position, it keeps carving cells out of the tessellation. The basic idea of our algorithm is to carve the tessellation along *layers of polygons*. We define the layering number $\zeta \in \aleph$ of a modeling primitive P in the following intuitive way. If we order each modeling primitive along each pixel by their positive† distance to the eye point, we define $\zeta(P)$ to be the smallest rank of P over all of the pixels to which it contributes. Clearly, $\zeta(P) = 1$, if, and only if, P is visible.

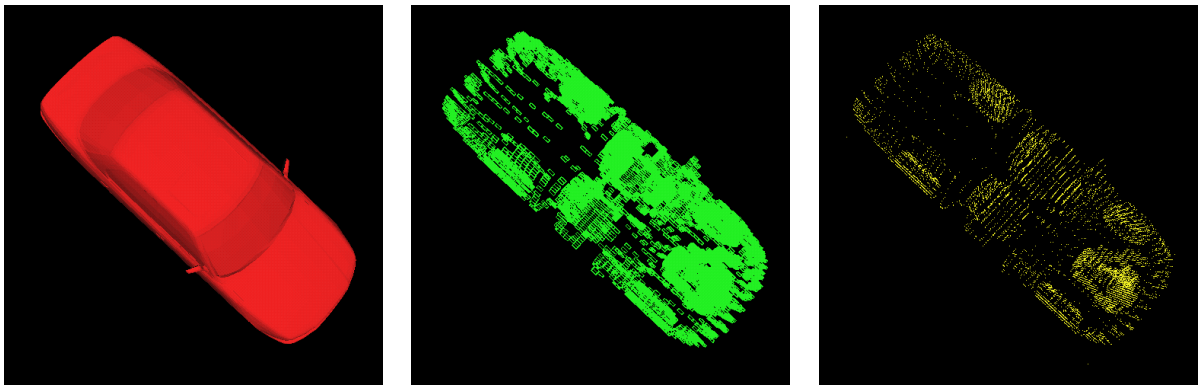
Finding the rank 1 primitives is equivalent to solving the visibility problem. Instead of solving this hard problem, the PLP algorithm uses simple heuristics. Our traversal algorithm attempts to project the modeling primitives by layers, that is, all primitives of rank 1, then 2 and so on. We do this by always projecting the cell in the front F (we call *the front*, the collection of cells that are immediate candidates for projection) which is least likely to be occluded according to their solidity values. Initially, the front is empty, and as cells are inserted, we estimate its accumulated solidity value to reflect its position during the traversal. (Cell solidity is defined below in Section 3.2). Every time a cell in the front is projected, all of the geometry assigned to it is rendered.

In Fig. 2, we can see a snapshot of our algorithm as it carves its way into space. The rendered geometry is shown in red and the cells in the front are shown in blue. Note how the algorithm has carved space around the body of the automobile.

There are several types of budgeting that can be applied to our technique, for example, a triangle count budget that can be

*Our cell-projection algorithm is different than the ones used in volume rendering in the following ways: (1) in volume rendering cells are usually projected in back-to-front order, while in our case, we project cells in *roughly* front-to-back order; (2) more importantly, we do not keep a strict depth-ordering of the cells during projection. This would be too restrictive – and expensive – for our purposes.

†Without loss of generality, assume P is in the view frustum.



(a) (b) (c)

Figure 3: Occupancy-based spatial tessellation algorithm. The input geometry, a car with an engine composed of over 160K triangles, is shown in (a). Using the vertices of the input geometry, we build an error-bounded octree, shown in (b). The centers of the leaf-nodes of the octree, shown in yellow in (c), are used as the vertices of our Delaunay Triangulation.

used to make it time-critical, and a ρ-budget, that, assuming ρ is (in some way) related to ζ leads to a more scene-independent budgeting scheme. For a given budget of k modeling primitives, let T_k be the set of primitives our traversal algorithm projects; we can also define T_ρ as the set of primitives projected for a particular ρ number. These sets, together with S, the set of all primitives, and V, the set of visible primitives, can be used to define several statistics that measure the overall effectiveness of our technique. The most relevant is the *visible coverage ratio* for a budget of k primitives, ε_k. This is the number of primitives in the visible set that we actually render, that is, $\varepsilon_k = \frac{|V \cap T_k|}{|V|}$. If $\varepsilon_k < 1$, we missed rendering some visible primitives. (ε_ρ can be defined analogously.)

PLP does not attempt to computed the visible set exactly. Instead, it combines a budget with its solidity-based polygon ordering. For a polygon budget of k, the best case scenario would be to have $\varepsilon_k = 1$. Of course, this would mean that for any view, PLP finds all of the visible polygons.

3.1 Occupancy-Based Spatial Tessellations

The underlying data structure used in our technique is a decomposition of the 3-space covered by the scene into disjoint cells. The characteristics we required in our spatial decomposition were:

(a) **Simple traversal characteristics** - must be easy and computationally inexpensive to walk from cell to cell.

(b) **Good projection properties** - depth-orderable from any viewpoint (with efficient, hopefully linear-time projection algorithms available); easy to estimate screen-space coverage.

(c) **Efficient space filler** - given an arbitrary set of geometry, it should be possible to "sample" the geometry adaptively, that is, with large cells in sparse areas, and smaller cells in dense areas.

(d) **Easy to build and efficient to store**.

We could have used any of a number of different spatial data structures, such as kd-trees or octrees, but we settled on a Delaunay triangulation, since it seems to most closely fill our needs. In making this decision, we were influenced by the work of Held *et al.* [14] on computing low-stabbing triangulations for collision detection; and Williams' MPVO [19], a linear-time algorithm for visibility ordering meshes mainly used for volume rendering.

In order to compute a spatial decomposition M, which adaptively samples the scene S, we use a very simple procedure that in effect just samples S with points; then constructs M as the Delaunay triangulation of the sample points; and finally assigns individual primitives in S to M. Fig. 3 shows our overall triangulation algorithm. Instead of accurately sampling the actual primitives (Fig. 3a), such as is done in [15], we simply construct an octree using only the original vertices (Fig. 3b); we limit the *depth* of the octree, which gives us a bound on the maximum complexity of our mesh; then we use the (randomly perturbed) center of the octree leaves as the vertices of our Delaunay triangulation(Fig. 3c). After M is built, we use a naive assignment of the primitives in S to M, by basically "scan-converting" the geometry into the mesh. Each cell $c_i \in M$, has a list of the primitives from S assigned to it. Each of these primitives is either completely contained in c_i, or it intersects one of its boundary faces. We use $|c_i|$, the number of primitives in cell i, in the algorithm that determine its solidity value. In a final pass over the data during preprocessing, we compute the maximum number of primitives in any cell, $\rho_{max} = \max_{i \in [1 \ldots |M|]} |c_i|$, to be used later as a scaling factor.

Remarks: (1) The resolution of the octree we use is very low. By default no leaf node has a side longer than 5% of the bounding box of S. This has shown to be quite satisfactory for all the experiments we have performed this far. (2) Even though primitives might be assigned to multiple cells of M (we use pointers to the actual primitives), the memory overhead has been negligible.

```
Algorithm RenderingLoop()
1.  while (empty(F ) != true)
2.      c = min(F )
3.      project(c)
4.      if ((reached_budget() == true)
5.              break;
6.      foreach n; n = cell_adjacent_to(c)
7.          if ((projected(n) == true)
8.                  continue;
9.          ρ = update_solidity(n, c)
10.         enqueue(n, ρ)
```

Figure 5: Skeleton of the *RenderingLoop* algorithm. Function *min*() returns the minimum element in the priority queue F. Function *project*(*c*) renders all the elements assigned to *c*; it also keeps counts on the number of primitives actually rendered. Function *reached_budget*() returns true if the we have already rendered *k* primitives. Function *cell_adjacent_to*(*c*) lists the cells adjacent to *c*. Function *projected*(*c*) returns true if cell *c* has already been projected. Function *update_solidity*(*n, c*) computes the updated solidity of cell *n*, based on the fact that *c* is one of its neighbors, and has just been projected. Function *enqueue*(*n*, ρ) places *n* in the queue with an solidity ρ. If *n* was already in the queue, this function will first remove it, and re-insert it with the updated solidity value. See text for more details on *update_solidity*().

3.2 Priority-Based Traversal Algorithm

Cell-projection algorithms [19, 17, 5] are implemented using queues or stacks, depending on the type of traversal (*i.e.*, depth-first versus breadth-first), and use some form of restrictive dependency among cells to ensure properties of the order of projection (*e.g.*, strict back-to-front).

Unfortunately such limited and strict projection strategies do not seem general enough to capture the notion of polygon layering, which we are using for visibility culling. In order for this to be feasible, we must be able to selectively stop (or at least delay) cell-projection around some areas, while continuing in others. In effect, we would like to project cells from M using a layering defined by the primitives in S. The intuitive notion we are trying to capture is as follows: if a cell c_i has been projected, and $|c_i| = \rho_{max}$, then the cells "behind" should wait until (at least) a corresponding "layer" of polygons in all other cells have been projected. Furthermore, in order to avoid any expensive image-based tests, we would prefer to achieve such a goal using only object-space tests.

In order to achieve this goal of capturing global solidity, we extend the cell-projection framework by replacing the fixed insertion/deletion strategy queue, with a metric-based queue (*i.e.*, a priority queue), so that we can control how elements get pushed and popped based on a metric we can define. We call this priority queue, F, the front. The complete traversal algorithm is shown in Fig. 5. In order to completely describe it, we need to provide details on solidity metrics and its update strategies.

Solidity. The notion of a cell solidity is the at the heart of our rendering algorithm shown in Fig. 5. At any given moment, cells are removed from the front (*i.e.*, priority queue F) in "solidity order", that is, the cells with the smallest solidity are projected before the ones with larger solidity. The solidity of a cell *c* used in the rendering algorithm is not an intrinsic property of the cell by itself. Instead we use a set of conditions to roughly estimate the visibility likelihood of a cell, and make sure that cells more likely to be visible get projected before less likely cells.

```
float function update_solidity(B, A)
    /* refer to Fig. 7 */
1.  ρ_acc = |A| / ρ_max + (v · n_B) * ρ_A
2.  if ((star_shaped(v, B) == false)
3.      ρ_acc = apply_penalty_factor(ρ_acc)
4.  return ρ_acc
```

Figure 6: Function *update_solidity*(). This function works as if transferring accumulated solidity from cell *A* into cell *B*. The maximum transfer happens if the new cell is well-aligned with the view direction, and in star-shaped position. If this is not the case, penalties will be incurred to the transfer as shown.

The notion of solidity is related to how difficult it is for the viewer to see a particular cell. The actual solidity value of a cell *c* is defined in terms of the solidity of the cells that intersect the closure of a segment from the cell *c* to the eye point. The heuristic we have chosen to define the solidity value of our cells is shown in Fig. 6.

We use several parameters in computing the solidity value.

- The normalized number of primitives inside *c*. This number, which is necessarily between 0 and 1, is $\frac{|c|}{\rho_{max}}$. The rationale is that the more primitives a cell owns, the more likely it is to obscure the cells behind it.

- Its position with respect to the viewpoint. We transfer a cell's solidity to a neighboring cell based on how orthogonal the face that it shared between cells is to the view direction (see Fig. 7).

  We also give preference to cells whose interiors are visible from the viewpoint. Here, we attempt to force the cells in the front to be as "star-shaped" as possible. The reason for this is to avoid projecting cells (with low solidity values) that are occluded by cells in the front (with high solidity values) which have not been projected yet. This is likely to happen as the front expands away from a bottleneck. See Fig. 1 and 4d. Actually, *forcing* the front to be star-shaped at every step of the way is too limiting a rule. This would basically produce a visibility ordering for the cells (such as the one computed in [17, 5]). Instead, we simply *penalize* the cells in the front that do not maintain this star-shaped quality. ‡

4 Implementation Details

We have implemented a system to experiment with the ideas presented in this paper. The code is a mix of Tcl/Tk and C++, and OpenGL for visualization. In all, we have about 6,000

‡Even without this constraint, the algorithm seems to work fine, as can be seen in Figures 1 and 4 and in our 2D demo shown in the video. Our 2D prototype does not make any use of star-shape constraints.

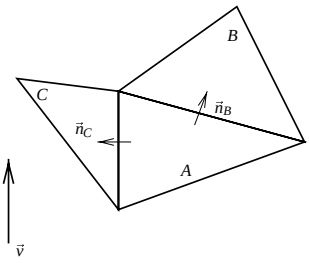


Figure 7: Solidity Transfer. After projecting cell *A*, the traversal algorithm will add cells *B* and *C* to the front. Based upon the current viewing direction, cell *B* will accumulate more solidity from *A* than will cell *C*, however, *C* will likely incur the non-star-shaped penalty. Refer to Fig. 6 for the transfer calculation.

lines of code. We briefly discuss the implementation of some of the main features of the code.

Data Structures. We need very simple data structures. Our current system only supports triangles for geometric primitives. Each triangle has pointers to its vertices, and a few flags, one of which is used to mark whether it has been rendered in the current scan. For the spatial tessellation, we represent each tetrahedron by pointer to its vertices, adjacency information is also required, as are a few flags used rendering purposes.

We keep the cells in the front in a priority queue. In our current implementation, we use an STL `set` to actually implement this data structure. Although simple and general, STL can add considerable overhead to an implementation. In our case, the number of cells in the front has been kept relatively small, and we have not noticed substantial slowdown due to STL.

Space Tessellation Code. We implemented our space tessellation code as three separate phases. First, vertices are inserted into a standard octree. We put a hard limit on the height of the octree based upon a user-defined error bound. Second, we compute a Delaunay triangulation of the (randomly perturbed) centers of the leaves of the bounding boxes. For this, we used `qhull`, software written at the Geometry Center, University of Minnesota. Even though our highly constrained input is bound to have several nasty degeneracies since all the points come from nodes of an octree, `qhull` had no problems handling it. Finally, we map triangles into the tetrahedra that contain them.

Rendering Loop Code. The rendering loop is basically a straightforward translation of the code in Fig. 5 into C++. Triangles are rendered very naively, one by one. We mark triangles as they are rendered, in order to avoid overdrawing triangles that get mapped to multiple cells. We also perform simple backface culling as well as view-frustum culling. We take no advantage of triangle-strips, vertex arrays, or other sophisticated OpenGL features.

Computing the "Exact" Visible Set. A number of benchmarking features are currently included in our implementation. One of the most useful is the computation of the actual "exact" visible set. We estimate V by using the well-know item buffer technique. In a nutshell, we color all the triangles with different colors, render them, and read the frame buffer back, recording which triangles contributed to the image rendered. After projection, all the rank-1 triangles have their "colors" imprinted into the frame buffer.

Centroid-Ordered Rendering. In order to have a basis for comparison, we implemented a simple ordering scheme based on sorting the polygons with respect to their centroid, and rendering them in that order up to the specified budget. Our implementation of this feature is naive, and tends to be slow, since it needs to touch every single triangle in S.

5 Experimental Results

We performed a series of experiments in order to determine the effectiveness of PLP's visibility estimation. Our experiments typically consist of recording a "flight path" consisting of several frames for a given dataset, then playback the path while varying the rendering algorithm used. We have three different strategies for rendering: (1) rendering every triangle in the scene at each frame, (2) centroid-based budgeting, or (3) PLP. During path playback, we also change the parameters when appropriate (*e.g.*, varying the polygon budget for PLP). Our primary benchmark machine is an IBM RS/6000 595 with a GXT800 graphics adapter. In all our experiments, rendering was performed using OpenGL with Z-buffer and lighting calculations turned on.

We report experimental results on two datasets:

City Model (CITY) The city model is composed of over 500K triangles (Fig. 8c). Each house has furniture inside, and while the number of triangles is large, the actual number of visible triangles per frame is quite small.

5 Car Body/Engine Model (5CBEM) This model has over 810K triangles (Fig. 9c). It is composed of five copies of a body and engine combination.

5.1 Preprocessing

Preprocessing involves computing an octree of the model, then computing a Delaunay triangulation of points defined by the octree (which is performed by calling `qhull`), and finally assigning the model geometric primitives to the spatial tessellation generated by `qhull`.

For the CITY model, preprocessing takes 70 seconds, and generated 25K tetrahedra. Representing each tetrahedron requires less than 100 bytes (assuming the cost of representing the vertices is amortized among several tetrahedra), leading to a memory overhead for the spatial tessellation on the order of 2.5MB. Another source of overhead comes from the fact that some triangles might be multiply assigned to tetrahedra. The average number of times a triangle is referenced is 1.80, costing 3.6 MB of memory (used for triangle pointers). The total memory overhead (on top of the original triangle lists) is 6.1 MB, while storing all the triangles alone (the minimal amount of memory necessary to render them) already costs 50 MB. So, PLP costs an extra 12% in memory overhead.

For the 5CBEM model, preprocessing took 135 seconds (also including the `qhull` time), and generated 60K tetrahedra. The average number of tetrahedra that points to a triangle

is 2.13, costing 14.7 MB of memory. The total memory overhead is 20 MB, and storing the triangles takes approximately 82 MB. So, PLP costs an extra 24% in memory overhead.

Since PLP's preprocessing only takes a few minutes, the preprocessing is performed online, when the user requests a given dataset. We also support offline preprocessing, by simply writing the spatial tessellation and the triangle assignment to a file.

5.2 Rendering

We performed several rendering experiments. During these experiments, the flight path used for the 5CBEM is composed of 200 frames. The flight path for the CITY has 160 frames. For each frame of the flight path, we computed the following statistics:

(1) the "exact" number of visible triangles in the frame.

(2) the number of visible triangles PLP was able to find for a given triangle budget. We varied the budget as follows: 1%, 2%, 5% and 10% of the number of triangles in the dataset.

(3) the number of visible triangles the centroid-based budgeting was able to find under a 10% budget.

(4) the number of "wrong" pixels generated by PLP.

(5) time (all times are reported in seconds) to render the whole scene.

(6) time PLP took to render a given frame.

(7) time the centroid-based budgeting took to render a given frame.

Several of the results (in particular, (1), (2), (3), (5), and (6)) are shown in Table 1, and Figs. 8a and 9a. The centroid rendering time (7) is mostly frame-independent, since the time is dominated by the sorting, which takes 6–7 seconds for the 5CBEM model, and 4–5 seconds for the CITY model. We collected the number of "wrong" pixels on a frame-by-frame basis. We report worst-case numbers. For the CITY model, PLP gets as many as 4% of the pixels wrong; for the 5CBEM model, this number goes up, and PLP misses as many as 12% of the pixels, in any given frame.

PLP seems to do quite a good job at finding visible triangles. In fact, looking at 8a, and 9a, we see a remarkable resemblance between the shape of the curve plotting the "exact" visible set, and the PLP's estimations. In fact, as the budget increases, the PLP curves seem to smoothly converge to the "exact" visible set curve. It is important to see that this is not a random phenomena. Notice how the centroid-based budgeting curve does not resemble the visible set curves. Clearly, there seems to be some relation between our heuristic visibility measure (captured by the solidity-based traversal), and actual visibility, which can not be captured by a technique that relies on distance alone.

Still, we would like PLP to do a better job at approximating the visible set. For this, it is interesting to see where it fails. In Figs 8d and 9d, we have 10%-budget images. Notice how PLP looses triangles in the back of the cars, (in Fig. 9d) since it estimates them to be occluded.

With respect to speed, PLP has very low overhead. For 5CBEM, at 1% we can render useful images at over 10 times the rate of the completely correct image, and for CITY, at 5% we can get 80% of the visible set, and still have four times faster rendering times.

Dataset/Budget	1%	2%	5 %	10%
City Model	51%	66%	80%	90%
5 Car Body/Engine Model	44%	55%	67%	76%

Table 1: Visible Coverage Ratio. The table summarizes ε_k for several budgets on two large models. The city model has 500K polygons, and the five car body/engine model has 810K polygons. For a budget of 1%, PLP is able to find over 40% of the visible polygons in either model.

Overall our experiments have shown that: (1) PLP can be applied to large data, without requiring large amounts of preprocessing; (2) PLP is able to find a large amount of visible geometry with a very low budget; (3) PLP is useful in practice, making it easier to inspect large objects, and in culling geometry that cannot be seen.

5.3 Video

The accompanying video shows the functionality of PLP on different datasets, and highlights both the occupancy-based spatial tessellation algorithm and solidity-based traversal algorithm. Video recording was performed live on an IBM Intellistation Z Pro running Microsoft Windows NT. In summary, video footage contains: (1) A demonstration of the 2D version of the PLP software on a set of line segments of varying depth complexity. The footage shows rendering animations for a budget of 500 line segments, as the user changes the eyepoint, and view direction. It is useful in understanding the behavior of PLP's traversal, and how it skips rendering occluded geometry. (2) A demonstration of the 3D version of the PLP software on the city model.

6 Conclusion and Future Work

In this paper, we proposed the Prioritized-Layered Projection algorithm. PLP renders geometry by carving out space along layers, while keeping track of the solidity of these layers as it goes along. PLP is very simple, requiring only a suitable tessellation of space where solidity can be computed (and is meaningful). The PLP rendering loop is a priority-based extension of the traversal used in depth-ordering cell projection algorithms developed originally for volume rendering.

We use PLP as our primary visibility-culling algorithm. Two things are most important to us. First, there is no offline preprocessing involved, that is, no need to simplify objects, pre-generate occluders, and so on. Second, its flexibility to adapt to multiple machines. In essence, in our application we were mostly interested in obtaining good image accuracy across a large number of machines with minimal time and space overheads. For several datasets, we can use PLP to render only 5% of a scene, and still be able to visualize over 80% of the visible polygons; if this is not enough, it is simple to adjust the budget for the desirable accuracy. A nice feature of PLP is that the visible set is stable, that is, the algorithm does not have major popping artifacts as it estimates the visible set from nearby viewpoints.

We see several other uses of PLP's rendering framework. A particularly intriguing one is to exploit PLP's ability to determine a large number of the visible polygons at low cost in terms of projected triangles (*e.g.*, PLP can find over 40% of the visible polygons while only projecting 1% of the original geometry) to improve the performance of other occlusion-culling techniques. For instance, at each frame, HOM [21] projects geometry to create occlusion maps. Instead of relying on preprocessed simplified geometry, HOM could rely on PLP's output for its set of occluders.

Although PLP's budget-based framework is useful as is, there are several interesting avenues for new research. We would like to understand better the relation of the solidity measure to the actual set of rendered polygons. Possibly, changing our solidity value computation can lead to even better performance. For example, accounting for front facing triangles in a given cell by considering their normals with respect to the view direction. The same is true for the mesh generation. Another class of open problems are related to further extensions in the front-update strategies. At this time, a single cell is placed in the front, after which the PLP traversal generates an ordering for all cells. We cut this tree by using a budget. It would be interesting to exploit the use of multiple initial seeds. Clearly, the best initial guess of what's visible, the easier it is to continue projecting visible polygons.

Acknowledgements

We would like to thank Dirk Bartz and Michael Meissner for the city model; Bengt-Olaf Schneider for suggesting adding star-shape constraints to the front; Fausto Bernadini, Paul Borrel, William Horn, and Gabriel Taubin for suggestions and help throughout the project; and the Geometry Center of the University of Minnesota for `qhull`.

References

[1] D. Bartz, M. Messner, and T. Huettner. Extending graphics hardware for occlusion queries in opengl. In *Proc. Workshop on Graphics Hardware '98*, pages 97–104, 1998.

[2] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974.

[3] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[4] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–253, 1998.

[5] J. Comba, J. Klosowski, N. Max, J. Mitchell, C. Silva, and P. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum (Eurographics '99 Proc.)*, To appear.

[6] S. Coorg and S. Teller. A spatially and temporally coherent object space visibility algorithm. Technical Report TM-546, Laboratory of Computer Science, MIT, 1996.

[7] S. Coorg and S. Teller. Temporally coherent conservative visibility. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 78–87, 1996.

[8] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[9] D. P. Dobkin and S. Teller. Computer graphics. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 42, pages 779–796. CRC Press LLC, Boca Raton, FL, 1997.

[10] S. E. Dorward. A survey of object-space hidden surface removal. *Internat. J. Comput. Geom. Appl.*, 4:325–362, 1994.

[11] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[12] T. Funkhouser and C. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[13] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[14] M. Held, J. Klosowski, and J. Mitchell. Collision detection for fly-throughs in virtual environments. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages V13–V14, 1996.

[15] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[16] J. Rohlf and J. Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In *Proceedings of SIGGRAPH '94)*, Annual Conference Series, pages 381–395, July 1994.

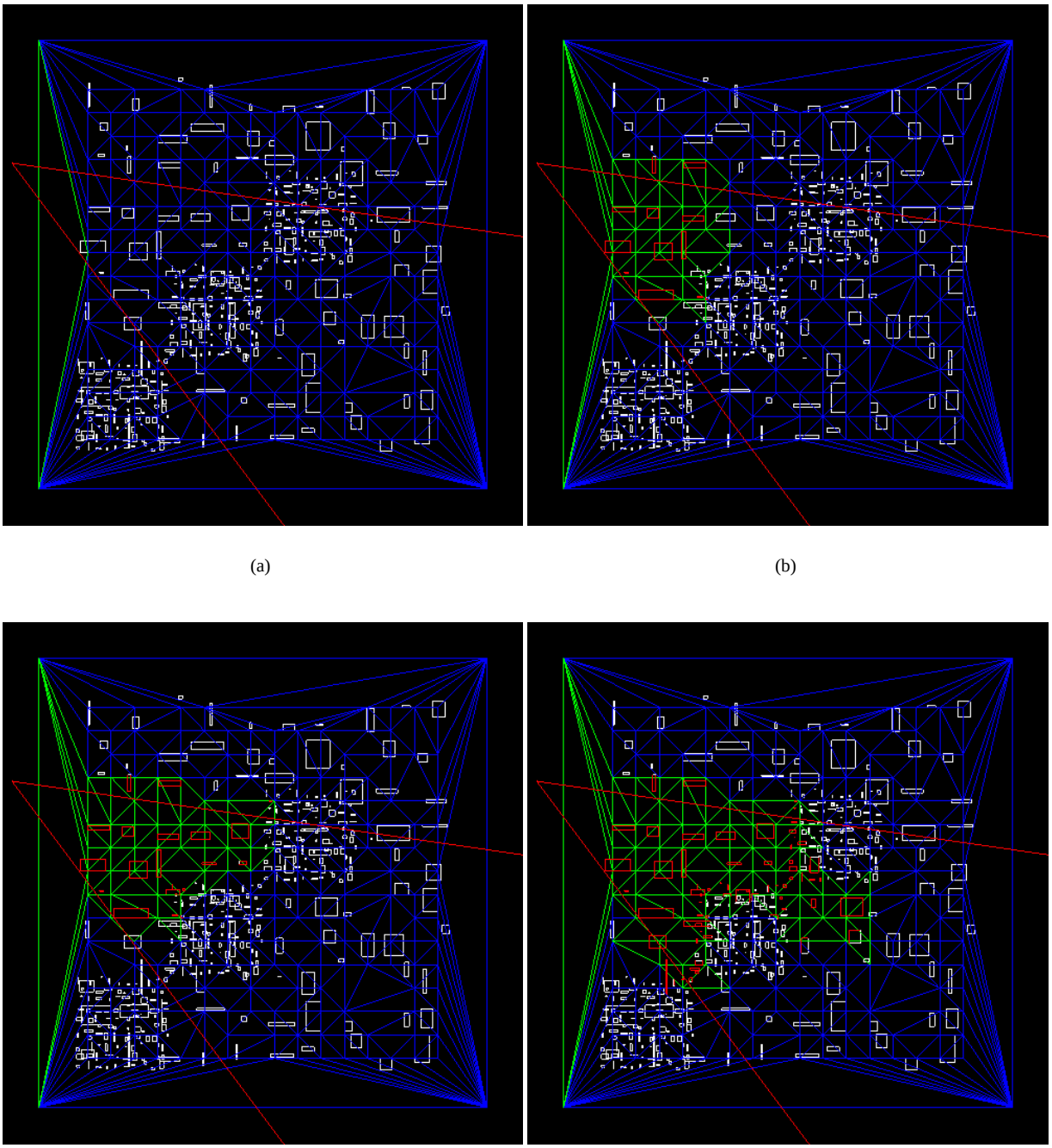
[17] C. Silva, J. Mitchell, and P. Williams. An interactive time visibility ordering algorithm for polyhedral cell complexes. In *Proc. ACM/IEEE Volume Visualization Symposium 98*, pages 87–94, November 1998.

[18] S. Teller and C. Séquin. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 61–69, July 1991.

[19] P. Williams. Visibility ordering meshes polyhedra. *ACM Transactions on Graphics*, 11(2), 1992.

[20] H. Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. PhD thesis, Department of Computer Science, University of North Carolina, Chappel Hill, 1998.

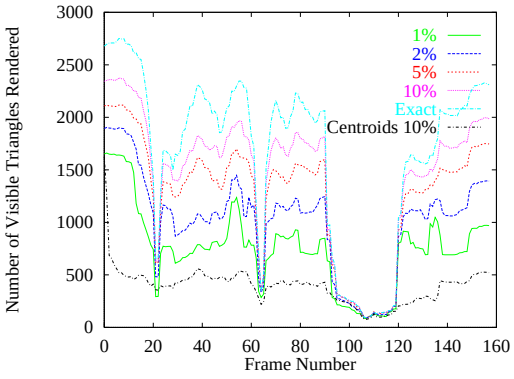
[21] H. Zhang, D. Manocha, T. Hudson, and K. Hoff III. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 77–88, August 1997.



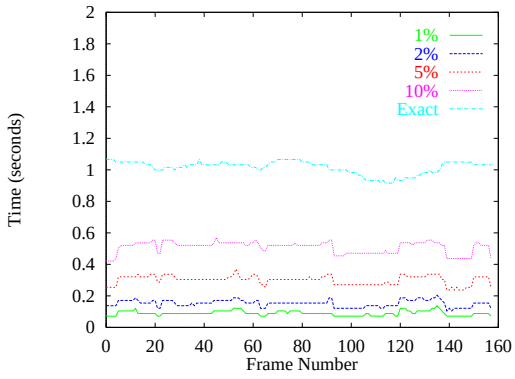
(a) (b)

(c) (d)

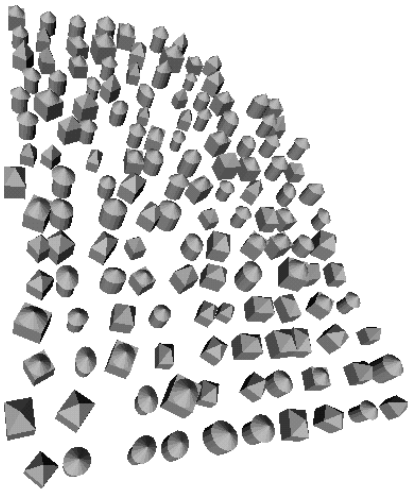
Figure 4: Priority-based traversal algorithm. In (a), the first cell, shown in green, gets projected. The algorithm continues to project cells based upon the solidity values. Note that the traversal, in going from (b) to (c), has delayed projecting those cells with a higher solidity value (*i.e.* a larger number of primitives) in the lower-left region of the view frustum. In (d), as the traversal continues, a higher priority is given to cells likely to have visible geometry, instead of projecting the ones inside of high-depth complexity regions. Note that the star-shaped criterion was not included in our 2D implementation.



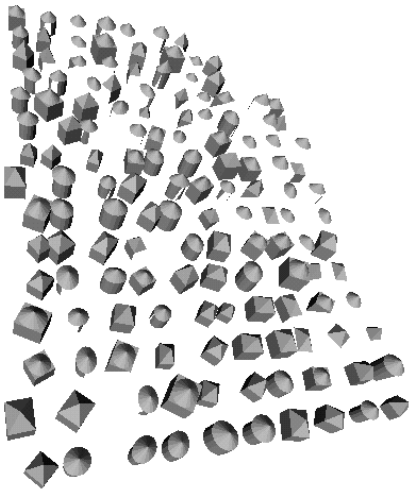
(a)



(b)

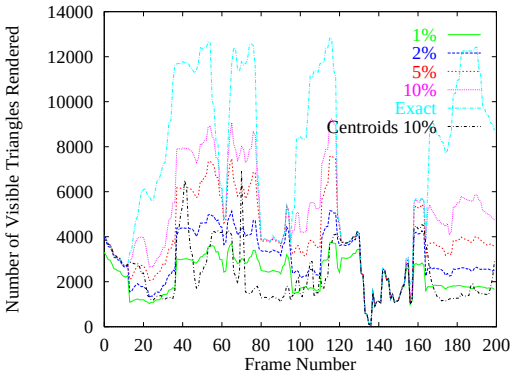


(c)

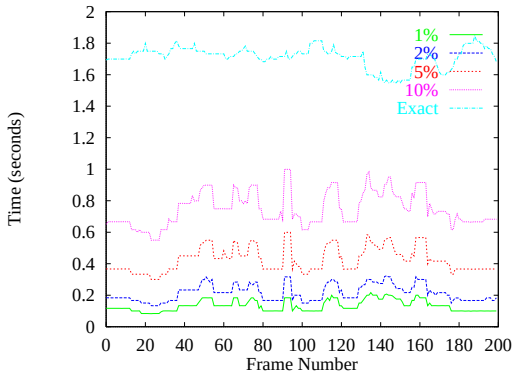


(d)

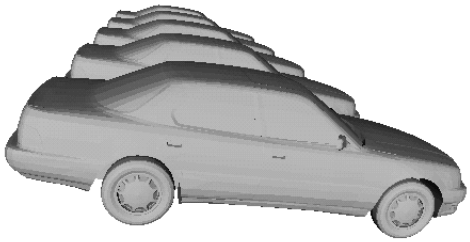
Figure 8: CITY results. (a) The top curve is the number of visible triangles for each given frame. The four bottom curves are the number of the visible triangles PLP finds with a given budget. Budgets of 1%, 2%, 5% and 10% are reported. (b) Rendering times in seconds for each curve shown in (a). (c) Image of all the visible triangles. (d) Image of the 10% PLP visible set.



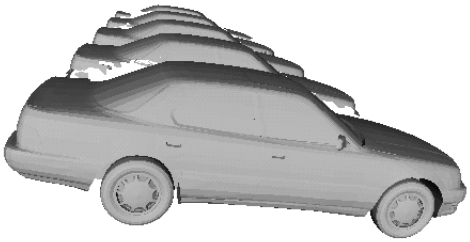
(a)



(b)



(c)



(d)

Figure 9: 5CBEM results. (a) The top curve is the number of visible triangles for each given frame. The four bottom curves are the number of the visible triangles PLP finds with a given budget. Budgets of 1%, 2%, 5% and 10% are reported. (b) Rendering times in seconds for each curve shown in (a). (c) Image of all the visible triangles. (d) Image of the 10% PLP visible set.