The Prioritized-Layered Projection Algorithm for Visible Set Estimation

James T. Klosowski and Cláudio T. Silva, *Member*, *IEEE*

Abstract—*Prioritized-Layered Projection* (PLP) is a technique for fast rendering of high depth complexity scenes. It works by *estimating* the visible polygons of a scene from a given viewpoint incrementally, one primitive at a time. It is not a conservative technique, instead PLP is suitable for the computation of partially correct images for use as part of time-critical rendering systems. From a very high level, PLP amounts to a modification of a simple view-frustum culling algorithm, however, it requires the computation of a special occupancy-based tessellation and the assignment to each cell of the tessellation a *solidity* value, which is used to compute a special ordering on how primitives get projected. In this paper, we detail the PLP algorithm, its main components, and implementation. We also provide experimental evidence of its performance, including results on two types of spatial tessellation (using octree- and Delaunay-based tessellations), and several datasets. We also discuss several extensions of our technique.

Index Terms—Visibility, time-critical rendering, occlusion culling, visible set, spatial tessellation.

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 geometry closer to the viewer. Quite possibly, the hardest part of the visibility-culling problem is to avoid rendering geometry that cannot be seen due to its being obscured by closer 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 smaller spatial cells where there are more geometric primitives, e.g., polygons; 2) in real-time, rendering is performed by traversing the cells in an order determined by their intrinsic solidity (likelihood of being occluded) and some other view-dependent information. As cells are projected, 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.

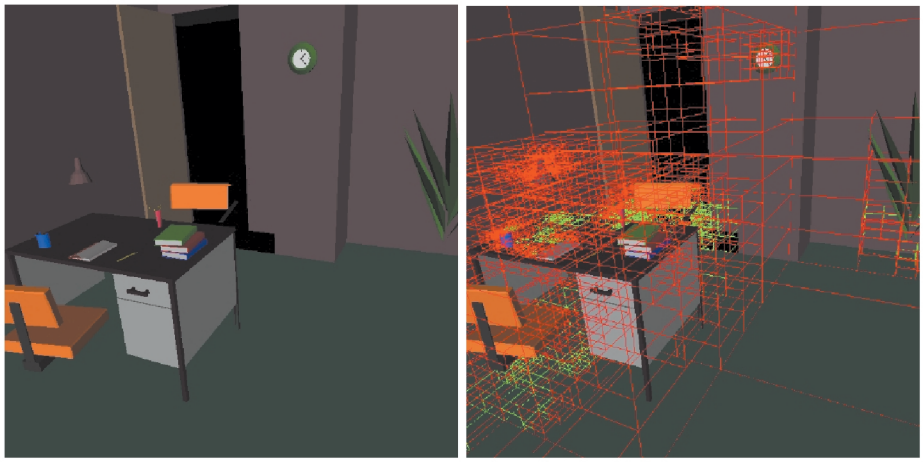
- *J.T. Klosowski is with the IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598. E-mail: jklosow@us.ibm.com.*
- *C.T. Silva is with AT&T Labs-Research, 180 Park Ave., PO Box 971, Florham Park, NJ 07932. E-mail: csilva@research.att.com*

Manuscript received 15 Mar. 2000; accepted 3 Apr. 2000.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 111484.

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 describe a few extensions and other avenues for future work. In Section 7, we conclude the paper with some final remarks.



(a) (b)

Fig. 1. The Prioritized-Layered Projection Algorithm. PLP attempts to prioritize the rendering of geometry along layers of occlusion. Cells that have been projected by the PLP algorithm are highlighted in red wireframe and their associated geometry is rendered, while cells that have not been projected are shown in green. Notice that the cells occluded by the desk are outlined in green, indicating that they have not been projected.

2 Preliminaries and Related Work

The visibility problem is defined in [9] as follows: Let the scene, $\mathcal{S}$, be composed of modeling primitives (e.g., triangles or spheres) $\mathcal{S} = \{\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n\}$ 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 that are visible, that is, connected to the eye point by a line segment that meets the closure of no other primitive. For a scene with $n = O(|\mathcal{S}|)$ 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 eye point 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 a 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 $\mathcal{V} \subset \mathcal{S}$ to be the set of modeling primitives which contribute to at least one pixel of the screen.

In computer graphics, visibility-culling research mainly focused on algorithms for computing conservative (hopefully tight) estimations of $\mathcal{V}$, 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 faces 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 $\mathcal{V}$ 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 role 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 $\mathcal{P}_i$ in front-to-back order and 2) keeping screen coverage information. Several efficiency issues are important for occlusion-culling algorithms:

1. They must operate under great time and space constraints since large amounts of geometry must be rendered in fractions of a second for real-time display.
2. 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 $\mathcal{P}_i$, should be avoided.
3. 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 $\mathcal{P}_0$ nor $\mathcal{P}_1$ obscures $\mathcal{P}_2$ by itself, but together they do cover $\mathcal{P}_2$. 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 [8], [9], [10]. For a survey of computer graphics work, see [28].

We very briefly survey some of the recent work more directly related to our technique. Hierarchical occlusion maps [29] 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, objects from a precomputed database are chosen to be occluders and used to cull geometry that cannot be seen. A closely related technique is the hierarchical Z-buffer [13].

A simple and effective hardware technique for improving the performance of the visibility computations with a Z-buffer has been proposed in [23]. The idea is to add a feedback loop in the hardware which is able to check if changes would have been made to the Z-buffer when scan-converting a given primitive.[1] This hardware makes it possible to check if a complex model is visible by first querying whether an enveloping primitive (often the bounding box of the object, but, in general, one can use any enclosing object, e.g., k-dop [16]), is visible and only rendering the complex object if the enclosing object is actually visible. Using this hardware, simple hierarchical techniques can be used to optimize rendering (see [17]). In [1], another extension of graphics hardware for occlusion-culling queries is proposed.

It is also possible to perform object-space visibility culling. One such technique, described in [26], 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 precomputing simple ray-shooting checks.

In [12], a constant-frame rendering system is described. This work uses the visibility-culling from [26]. 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 [21], 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 conservatively overestimating $\mathcal{V}$, our algorithm works on a budget. At each frame, the user can provide a maximum number of primitives to be rendered, i.e., a polygon budget, 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.

1. In OpenGL, the technique is implemented by adding a proprietary extension that can be enabled when queries are being performed.

Our projection strategy is completely object-space based, and resembles[2] 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 cells is generated in such a way as to roughly keep a uniform density of primitives per cell. Our sampling leads to large cells in unpopulated areas and small cells 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.

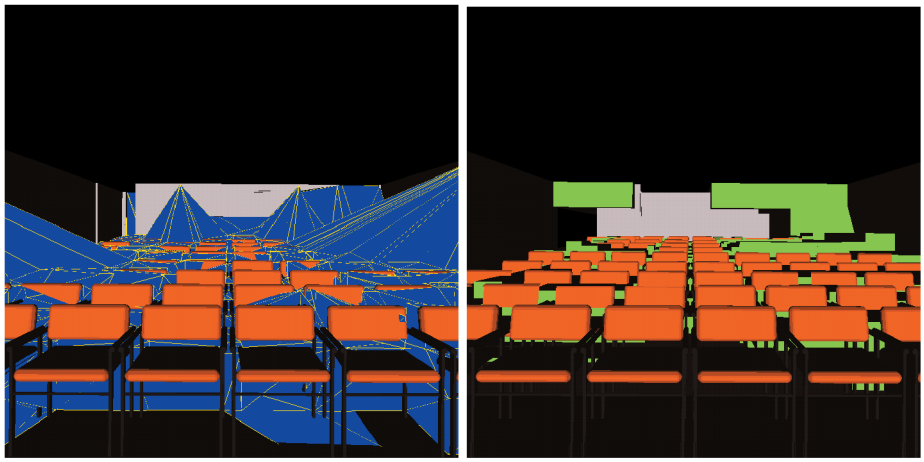
Generating such a space tessellation is not a very expensive step, e.g., taking only a minute or two minutes for a scene composed of one million triangles and, for several large datasets, can even be performed as part of the data input process. Of course, for truly large datasets, we highly recommend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

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 $\mathcal{P}$ in the following intuitive way: If we order each modeling primitive along each pixel by their positive[3]) distance to the eye point, we define $\zeta(\mathcal{P})$ to be the smallest rank of $\mathcal{P}$ over all of the pixels to which it contributes. Clearly, $\zeta(\mathcal{P}) = 1$ if, and only if, $\mathcal{P}$ 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 $\mathcal{F}$ (we call *the front*, the collection of cells that are immediate candidates for projection) which is least likely to be occluded according to its 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 see a snapshot of our algorithm for each of the spatial tessellations that we have implemented. The cells which have not been projected in the Delaunay

2. 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.

3. Without loss of generality, assume $\mathcal{P}$ is in the view frustum.



(a) (b)

Fig. 2. The input geometry is a model of a seminar room. Snapshots of the PLP algorithm highlight the spatial tessellations that are used. The cells which have not been projected in the (a) Delaunay triangulation and (b) the octree are highlighted in blue and green, respectively. At this point in the algorithm, the geometry associated with the projected cells has been rendered.

triangulation Fig. 2a and the octree Fig. 2b are highlighted in blue and green, respectively.

There are several types of budgeting that can be applied to our technique, for example, a triangle-count budget can be used to make it time-critical. For a given budget of k modeling primitives, let $\mathcal{T}_k$ be the set of primitives our traversal algorithm projects. This set, together with $\mathcal{S}$, the set of all primitives, and $\mathcal{V}$, the set of visible primitives, can be used to define several statistics that measure the overall effectiveness of our technique. One relevant statistic 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, $\varepsilon_k = \frac{|\mathcal{V} \cap \mathcal{T}_k|}{|\mathcal{V}|}$. If $\varepsilon_k < 1$, we missed rendering some visible primitives.

PLP does not attempt to compute 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 PLP finds all of the visible polygons.

In addition to the visible coverage ratio ε_k, another important statistic is the number of incorrect pixels in the image produced by the PLP technique. This provides a measure as to how closely the PLP image represents the exact image produced by rendering all of the primitives.

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:

1. **Simple traversal characteristics**—must be easy and computationally inexpensive to walk from cell to cell.
2. **Good projection properties**—depth-orderable from any viewpoint (with efficient, hopefully linear-time projection algorithms available); easy to estimate screen-space coverage.
3. **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.
4. **Easy to build and efficient to store**.

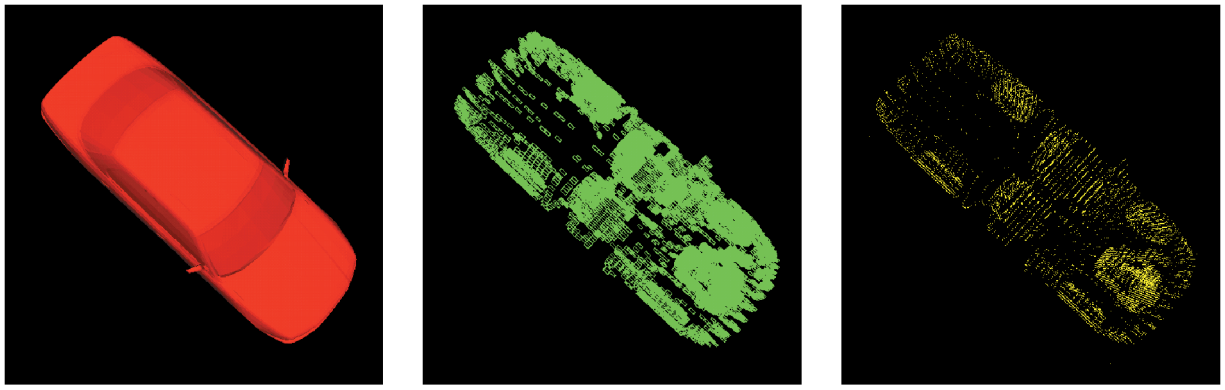
It is possible to use any of a number of different spatial data structures, such as kd-trees, octrees, or Delaunay triangulations. The particular use of one kind of spatial tessellation may be related to the specific dataset characteristics, although our experiments have shown that the technique works with at least two types of tessellations (octrees and Delaunay triangulations).

Overall, it seems that using low-stabbing triangulations, such as those used by Held et al. [14] (see also Mitchell et al. [18], [19] for theoretical properties of such triangulations), which are also depth-sortable (see [27], [25], [5]) are a good choice for occupancy-based tessellations. The main reason for this is that, given any path in space, these triangulations tend to minimize the traversal cost, allowing PLP to efficiently find the visible surfaces.

In order to actually compute a spatial decomposition $\mathcal{M}$ which adaptively samples the scene, we use a very simple procedure, explained in Section 4. After $\mathcal{M}$ is built, we use a naive assignment of the primitives in $\mathcal{S}$ to $\mathcal{M}$ by basically scan-converting the geometry into the mesh. Each cell $c_i \in \mathcal{M}$, has a list of the primitives from $\mathcal{S}$ 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 c_i, in the algorithm that determines the solidity values of c_i's neighboring cells. In a final pass over the data during preprocessing, we compute the maximum number of primitives in any cell, $\rho_{max} = \max_{i \in [1 \ldots |\mathcal{M}|]} |c_i|$, to be used later as a scaling factor.

3.2 Priority-Based Traversal Algorithm

Cell-projection algorithms [27], [25], [5] are implemented using queues or stacks, depending on the type of traversal (e.g., 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).



(a) (b) (c)

Fig. 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.

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 $\mathcal{M}$ using a layering defined by the primitives in $\mathcal{S}$. 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)

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

Fig. 4. Skeleton of the *RenderingLoop* algorithm. Function $min(\mathcal{F})$ returns the minimum element in the priority queue $\mathcal{F}$. Function $project(c)$ renders all the elements assigned to cell c; it also counts the number of primitives actually rendered. Function *reached_budget*() returns **true** if we have already rendered k primitives. Function *cell_adjacent_to*(c) lists the cells adjacent to c. Function *projected*(n) returns **true** if cell n 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 a solidity ρ. If n was already in the queue, this function will first remove it and reinsert it with the updated solidity value. See text for more details on *update_solidity*().

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, $\mathcal{F}$, the front. The complete traversal algorithm is shown in Fig. 4. In order to completely describe it, we need to provide details on solidity metrics and its update strategies.

Solidity. The notion of a cell's solidity is at the heart of our rendering algorithm shown in Fig. 4. At any given moment, cells are removed from the front (i.e., priority queue $\mathcal{F}$) in *solidity order*, that is, the cells with the smallest solidity are projected before the ones with larger solidity. The solidity of a cell B 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 the cell and make sure that cells more likely to be visible get projected before cells that are less likely to be visible.

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 B is defined in terms of the solidity of the cells that intersect the closure of a segment from the cell B to the eye point. The heuristic we have chosen to define the solidity value of our cells is shown in Fig. 5.

We use several parameters in computing the solidity value.

- The normalized number of primitives inside cell A, the neighboring cell (of cell B) that was just projected. This number, which is necessarily between 0 and 1, is $\frac{|A|}{\rho_{max}}$. The rationale is that the more primitives cell A contains, 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

```
float function update_solidity(B, A)
    /* refer to Fig. 6 */
1.  ρ_B = |A| / ρ_max + (v · n_B) * ρ_A
2.  if ((star_shaped(v, B) == false)
3.      ρ_B = apply_penalty_factor(ρ_B)
4.  return ρ_B
```

Fig. 5. Function *update_solidity*(). This function works as if transferring accumulated solidity from cell A into cell B. ρ_B is the solidity value to be computed for cell B. $|A|$ is the number of primitives in cell A. ρ_{max} is the maximum number of primitives in any cell. $\vec{n_B}$ is the normal of the face shared by cells A and B. ρ_A is the accumulated solidity value for cell A. The maximum transfer happens if the new cell is well-aligned with the view direction $\vec{v}$ and in star-shaped position. If this is not the case, penalties will be incurred to the transfer.

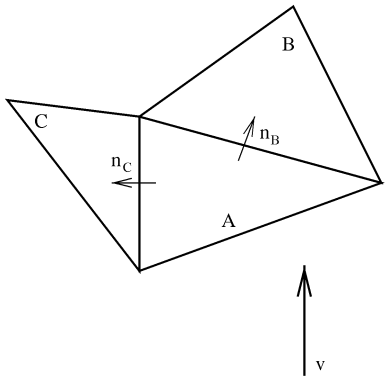


Fig. 6. Solidity Transfer. After projecting cell A, the traversal algorithm will add cells B and C to the front. Based upon the current viewing direction $\vec{v}$, cell B will accumulate more solidity from A than will cell C, however, C will likely incur the non-star-shaped penalty. $\vec{n_B}$ and $\vec{n_C}$ are the (respective) normals of the faces shared by the cell A's neighboring cells. Refer to Fig. 5 for the transfer calculation.

on how orthogonal the face that is shared between cells is to the view direction $\vec{v}$ (see Fig. 6).

We also give preference to neighboring cells that are star-shaped [8] with respect to the viewpoint and the shared face. That is, we attempt to force the cells in the front to have their interior, e.g., their center point, visible from the viewpoint along a ray that passes through the face shared by the two cells. 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 an area in the scene where two densely occupied regions are nearby; we refer to such an area as a bottleneck. Examples of such areas can easily be seen in Fig. 7, which highlights our 2D prototype implementation. 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 [25], [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 written in C++, with a mixture of Tcl/Tk and OpenGL for visualization. In order to access OpenGL functionality in a Tcl/Tk application, we use Togl [20]. In all, we have about to 10,000 lines of code. The code is very portable, and the exact same source code compiles and runs under IBM AIX, SGI IRIX, Linux, and Microsoft Windows NT. See Fig. 8 for a screen shot of our graphical user interface.

One of the reasons for the large amount of code actually comes from our flexible benchmarking capabilities. Among other functionality, our system is able to record and replay scene paths; automatically compute several different statistics about each frame as they are rendered (e.g., number of visible triangles, incorrect pixels); compute PLP traversals step-by-step; and "freeze" in the middle of a traversal to allow for the study of the traversal properties from different viewpoints.

Here is a brief discussion of some of the important aspects of our implementation:

4.1 Rendering Data Structures

At this time, the main rendering primitive in our system is the triangle. In general, we accept and use "triangle soups" as our input. For each triangle, we save pointers to its vertices (which include color information) and a few flags, one of which is used to mark whether it has been rendered in the current traversal. At this point, we do not make any use of the fact that triangles are part of larger objects. Triangles are assigned to cells and their renderings are triggered by the actual rendering of a cell. Although triangles can (in general) be assigned to more than one cell, they will only be rendered once per frame. A cell might get partially rendered in case the triangle budget is reached while attempting to render that particular cell.

4.2 Traversal Data Structures and Rendering Loop

During rendering, cells need to be kept in a priority queue. In our current implementation, we use an STL `set` to actually implement this data structure. We use an object-oriented design, which makes it easy for the traversal rendering code to support different underlying spatial tessellations. For instance, at this time, we support both octree and Delaunay-based tessellations. Since we are using C++, it is quite simple to do this. The following methods need to be supported by any cell data structure (this list only includes methods needed for the rendering traversal; other methods are needed for initialization and triangle assignment, and also for benchmarking):

- `calculateInitialSolidityValues(int` ρ_{max}`)`—Uses the techniques presented in Section 3.2 for computing the initial solidity.
- `getSolidity(), setSolidity(), getOriginalSolidity()`—Uses the techniques presented in Section 3.2 for updating the solidity values during traversal. Solidity updates need to be adjusted for different kinds of spatial tessellations.

We use these functions to define a comparator class (`less< >`) that can be used by the STL `set` to

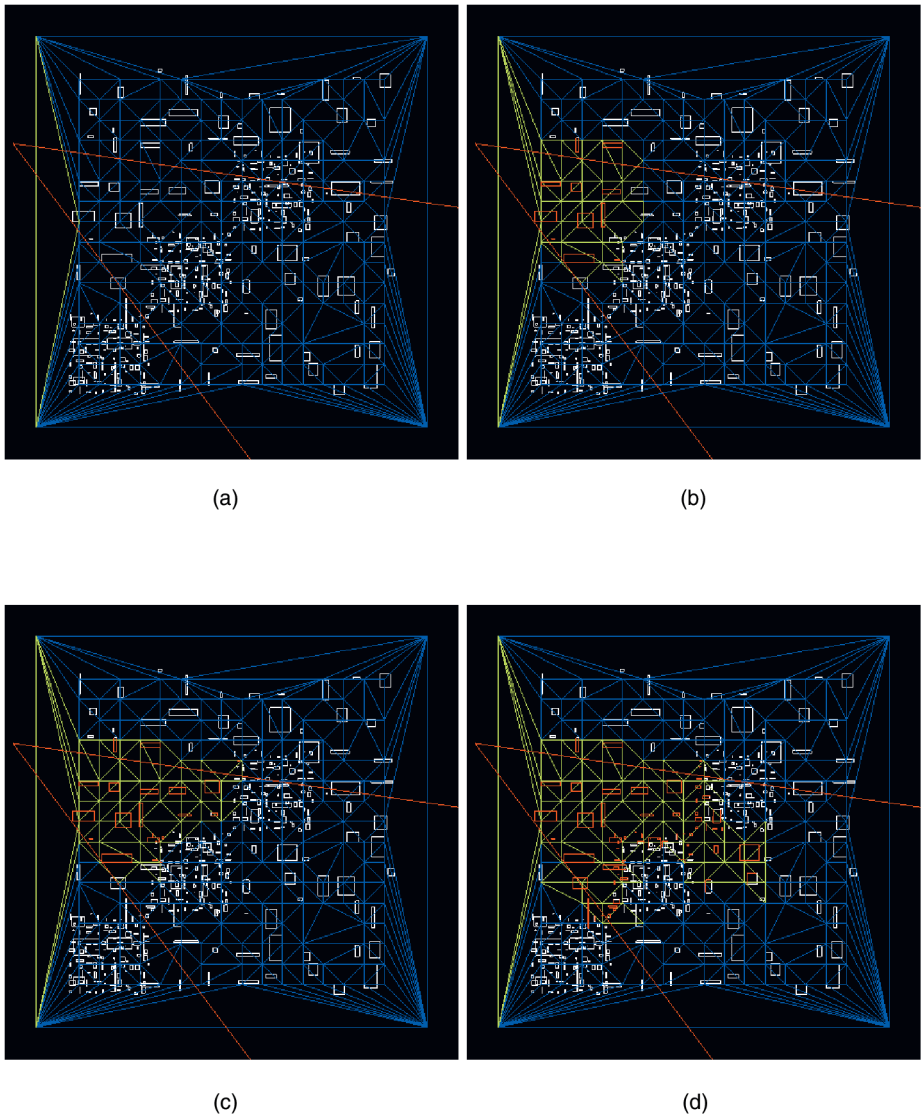


Fig. 7. Priority-based traversal algorithm. In (a), the first cell, shown in green, gets projected. The algorithm continues to project cells based upon the solidity values. Note that the traversal, in going from (b) to (c), has delayed projecting those cells with a higher solidity value (i.e., those cells less likely to be visible) 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.

sort the different cells. Each cell also has an internal timestamp, which is used to guarantee a first-in first-out behavior when there are ties with respect to the solidity values.

- `findCell(float vp[3])`—Find the cell that contains the viewpoint or returns that the viewpoint is outside the convex hull of the tessellation. (In order to jump start the traversal algorithm when the viewpoint is outside the tessellation, we use the cell that is closest to the viewpoint.)
- `getGeometry_VEC()`—Returns a reference to the list of primitives inside this cell.
- `getNeighbors_VEC()`—Returns a reference to the list of neighbors of this cell. (We also save the direction which identifies the face the two cells share. This is used to perform the solidity update on the neighboring cells.)

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.

The rendering loop is basically a straightforward translation of the code in Fig. 4 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.

4.3 Space Tessellation Code

This is quite possibly the most complicated part of our implementation and it consists of two parts, one for each of the two spatial tessellations we support. There is a certain amount of shared code since it is always necessary to first

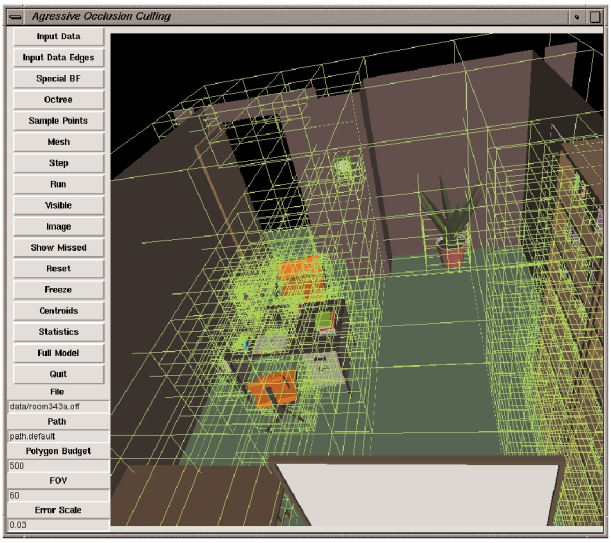


Fig. 8. Snapshot of our Tcl/Tk graphical user interface.

compute an adaptive sampling from the scene, for which we use a simple octree.

In more detail, in order to compute a spatial decomposition $\mathcal{M}$, which adaptively samples the scene $\mathcal{S}$, we use a very simple procedure that, in effect, just samples $\mathcal{S}$ with points, then (optionally) constructs $\mathcal{M}$ as the Delaunay triangulation of the sample points, and, finally, assigns individual primitives in $\mathcal{S}$ to $\mathcal{M}$. 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 size of the octree leaves, which gives us a bound on the maximum complexity of our mesh.[4] Note that, at this point, we do not have a space partitioning where we can run PLP; instead, the octree provides a hierarchical representation of space (i.e., the nodes of the octree overlap and are nested).

Once the octree has been computed with the vertex samples, we can generate two different types of subdivisions:

- **Delaunay triangulation**—We can use the (randomly perturbed) center of the octree leaves as the vertices of our Delaunay triangulation (Fig. 3c).

  For this, we used `qhull`, software written at the Geometry Center, University of Minnesota. Our highly constrained input is bound to have several degeneracies as all the points come from nodes of an octree, therefore we randomly perturbed these points and `qhull` had no problems handling them.

  After $\mathcal{M}$ is built, we use a naive assignment of the primitives in $\mathcal{S}$ to $\mathcal{M}$ by basically scan-converting the geometry into the mesh. Each cell $c_i \in \mathcal{M}$ has a list of the primitives from $\mathcal{S}$ assigned to it. Each of these primitives is either completely contained in c_i, or it intersects one of its boundary faces.

  Each tetrahedron is represented by pointers to its vertices. Adjacency information is also required, as are a few flags for rendering purposes.

4. 1) The resolution of the octree we use is very modest. By default, once an octree node has a side shorter than 5 percent of the length of the bounding box of $\mathcal{S}$, it is considered a leaf node. This has been shown to be quite satisfactory for all the experiments we have performed thus far. 2) Even though primitives might be assigned to multiple cells of $\mathcal{M}$ (we use pointers to the actual primitives), the memory overhead has been negligible. See Section 5.1.

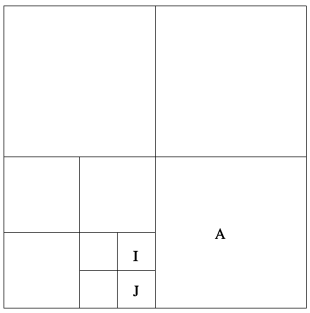


Fig. 9. Finding neighbors within the octree.

- **Octree**—Since we have already built an octree, it is obvious that we can use the same octree to compute a subdivision of space for PLP. Conceptually, this is quite simple since the leaves of the octree are guaranteed to form a subdivision of space. All that is really needed is to compute neighborhood information in the octree, for instance, looking at Fig. 9, we need to find that node A is a "face" neighbor of node I and J and vice-versa.

  Samet [22] describes several techniques for neighbor finding. The basic idea in finding the "face" neighbor of a node is to ascend the octree until the nearest common ancestor is found and to descend the octree in search of the neighbor node. In descending the octree, one needs to reflect the path taken while going up (for details, see Table 3.11 in [22]).

  One shortcoming with the technique as described in [22] is that it is only possible to find a neighbor at the same level or above (that is, it is possible to find that A is the "right" neighbor of I, but it is not possible to go the other way). A simple fix is to traverse the tree from the bottom to the top and allow the deeper nodes (e.g., I) to complete the neighborhood lists of nodes up in the tree (e.g., A).

Regardless of the technique used for subdivision, for the solidity calculations, we use $|c_i|$, the number of primitives in cell c_i, in the algorithm that determines the solidity values of c_i's neighboring cells. In a final pass over the data during preprocessing, we compute the maximum number of primitives in any cell, $\rho_{max} = \max_{i \in [1 \ldots |\mathcal{M}|]} |c_i|$, to be used later as a scaling factor.

4.4 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 $\mathcal{V}$ by using the well-known 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 rendering, all the rank-1 triangles have their colors imprinted into the frame buffer.

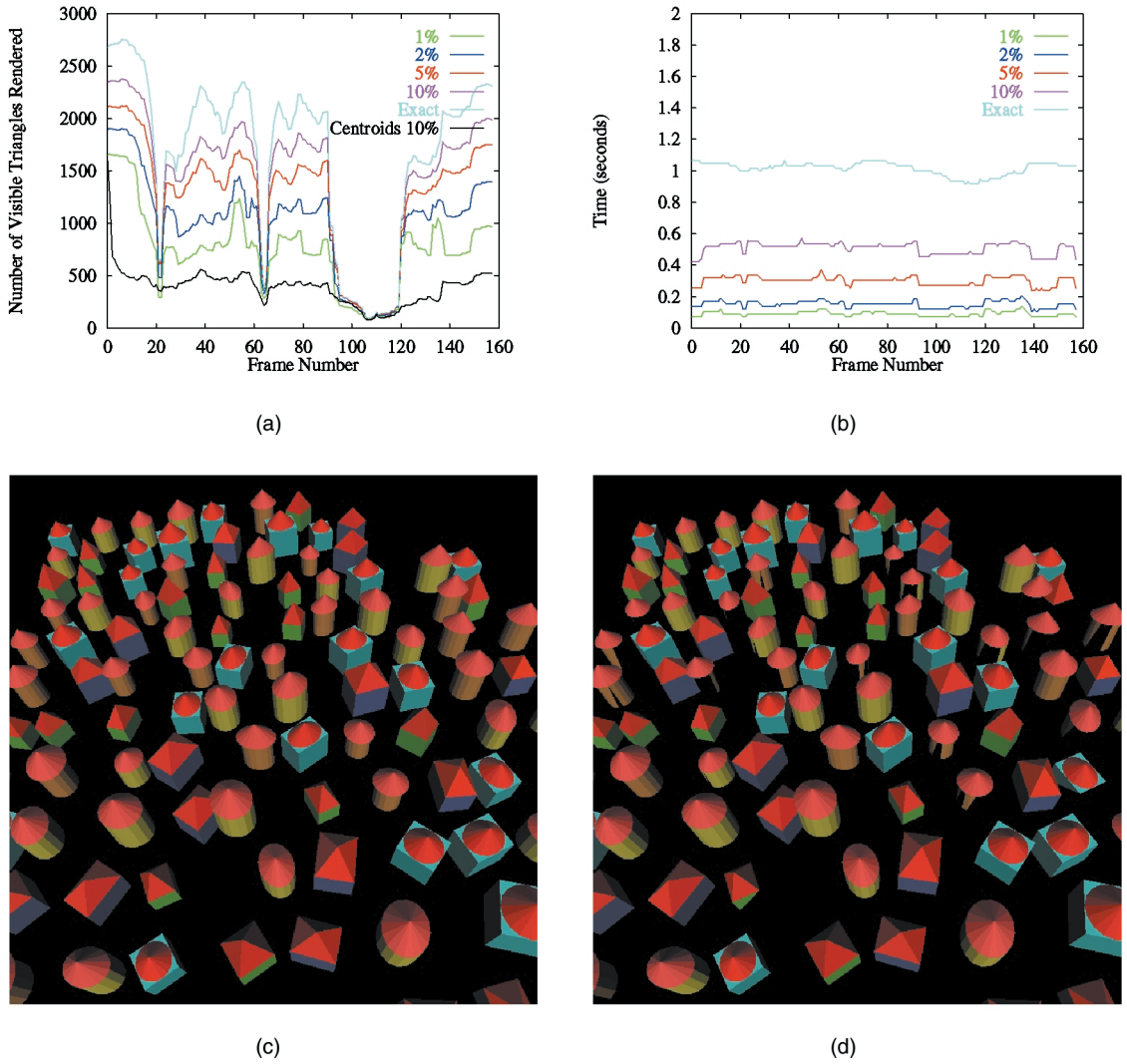


Fig. 10. CITY results. (a) The top curve, labeled Exact, is the number of visible triangles for each given frame. The next four curves are the number of visible triangles PLP finds with a given budget. From top to bottom, budgets of 10 percent, 5 percent, 2 percent, and 1 percent are reported. The bottom curve is the number of visible triangles that the centroid sorting algorithm finds. (b) Rendering times in seconds for each curve shown in (a), with the exception of the centroid sorting algorithm, which required 4-5 seconds per frame. (c) Image of all the visible triangles. (d) Image of the 10 percent PLP visible set.

4.5 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 tends to be slow for large datasets, as it needs to sort all of the triangles in $\mathcal{S}$ at each frame.

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 playing back the path while varying the rendering algorithm used. We have four different strategies for rendering: 1) rendering every triangle in the scene at each frame, 2) centroid-based budgeting, 3) PLP with octree-based tessellation, and 4) PLP with Delaunay triangulation. 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. In addition, all three algorithms perform view-frustum and backface culling to avoid rendering those triangles that clearly will not contribute to the final image. Thus, any benefits provided by PLP will be on top of the benefits provided by traditional culling techniques.

We report experimental results on three datasets:

Room 306 of the Berkeley SODA Hall (ROOM). This model has approximately 45K triangles (see Figs. 13 and 14) and consists of a number of chairs in what appears to be a reasonably large seminar room. This is a difficult model to perform visibility culling on since the number of visible triangles along a path varies quite a bit with respect to the total size of the dataset, in fact, in the path

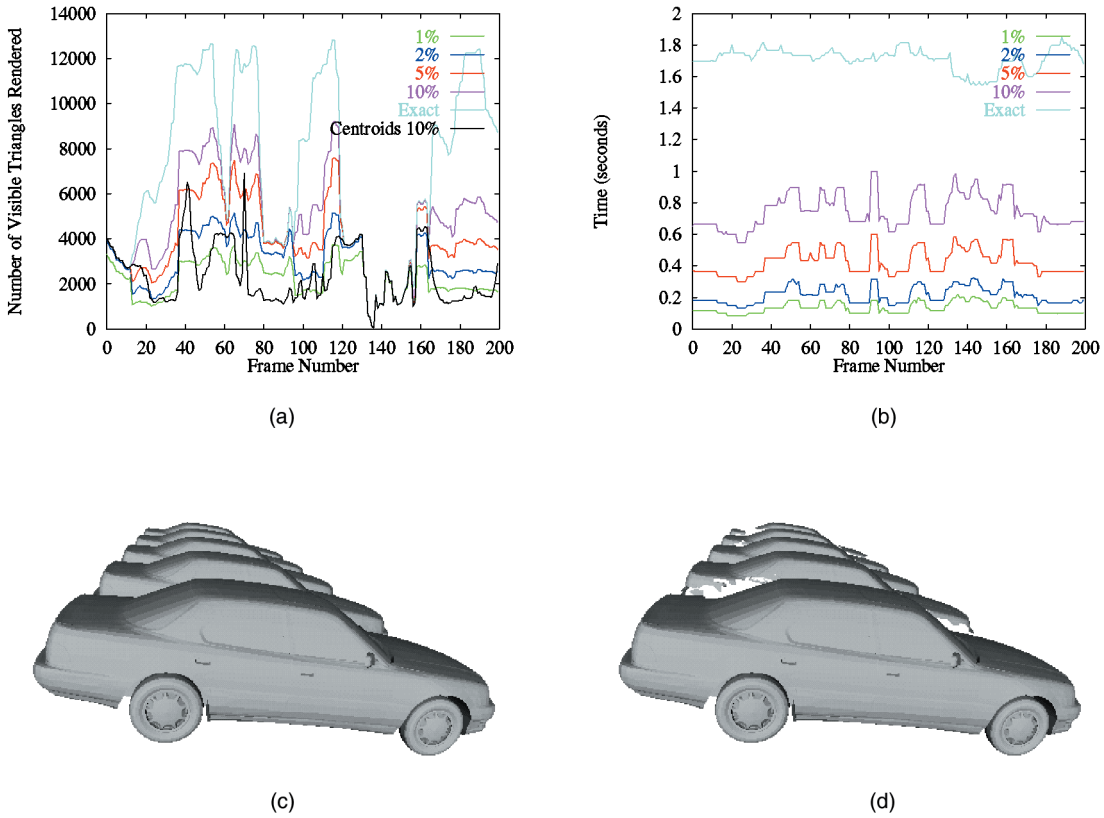


Fig. 11. 5CBEM results. (a) The top curve, labeled Exact, is the number of visible triangles for each given frame. The next four curves are the number of the visible triangles PLP finds with a given budget. From top to bottom, budgets of 10 percent, 5 percent, 2 percent, and 1 percent are reported. The bottom curve is the number of visible triangles that the centroid sorting algorithm finds. (b) Rendering times in seconds for each curve shown in (a), with the exception of the centroid sorting algorithm, which required 6-7 seconds per frame. (c) Image of all the visible triangles. (d) Image of the 10 percent PLP visible set.

we use, this number ranged from 1 percent to 20 percent of the total number of triangles.

City Model (CITY). The city model is composed of over 500K triangles (Fig. 10c). 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. 11c). It is composed of five copies of an automobile body and engine.

5.1 Preprocessing

Preprocessing involves computing an octree of the model, then (optionally) 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 took 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 percent 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 percent 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. The flight path for the ROOM has 235 frames. For each frame of the flight path, we computed the following statistics:

1. The exact number of visible triangles in the frame, estimated using the item-buffer technique.

TABLE 1
Visible Coverage Ratio

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

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 percent, PLP is able to find over 40 percent of the visible polygons in either model.

2. The number of visible triangles PLP was able to find for a given triangle budget. We varied the budget as follows: 1 percent, 2 percent, 5 percent, and 10 percent 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 percent 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. 10 and 11, which show PLP's overall performance and how it compares to the centroid-sorting based approach. 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 (4) on a frame-by-frame basis. We report worst-case numbers. For the CITY model, PLP gets as many as 4 percent of the pixels wrong; for the 5CBEM model, this number goes up and PLP misses as many as 12 percent of the pixels in any given frame.

The other figures focus on highlighting specific features of our technique, and compare the octree and Delaunay-based tessellations.

5.2.1 Speed and Accuracy Comparisons on the CITY Model

Fig. 12c shows the rendering times of the different algorithms and compares them with the rendering of the entire model geometry. For a budget of 10 percent, the Delaunay triangulation was over two times faster, while the

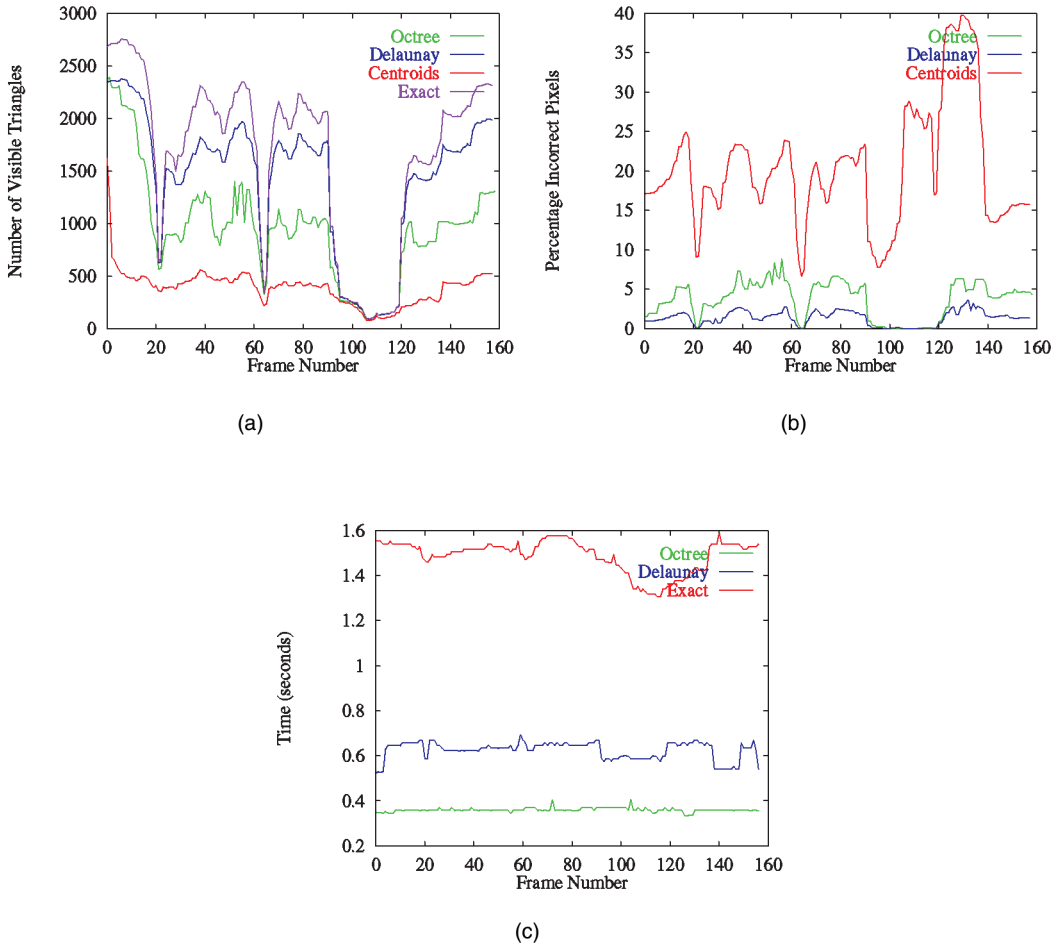


Fig. 12. This figure illustrates the quantitative differences among the different rendering techniques for each frame of the CITY path. In each plot, we report results for each rendering technique (centroid, octree-based PLP, and Delaunay-based PLP, respectively). In (a), we show the percentage of the visible polygons that each technique was able to find. In (b), we show the number of incorrect pixels in the images computed with each technique.

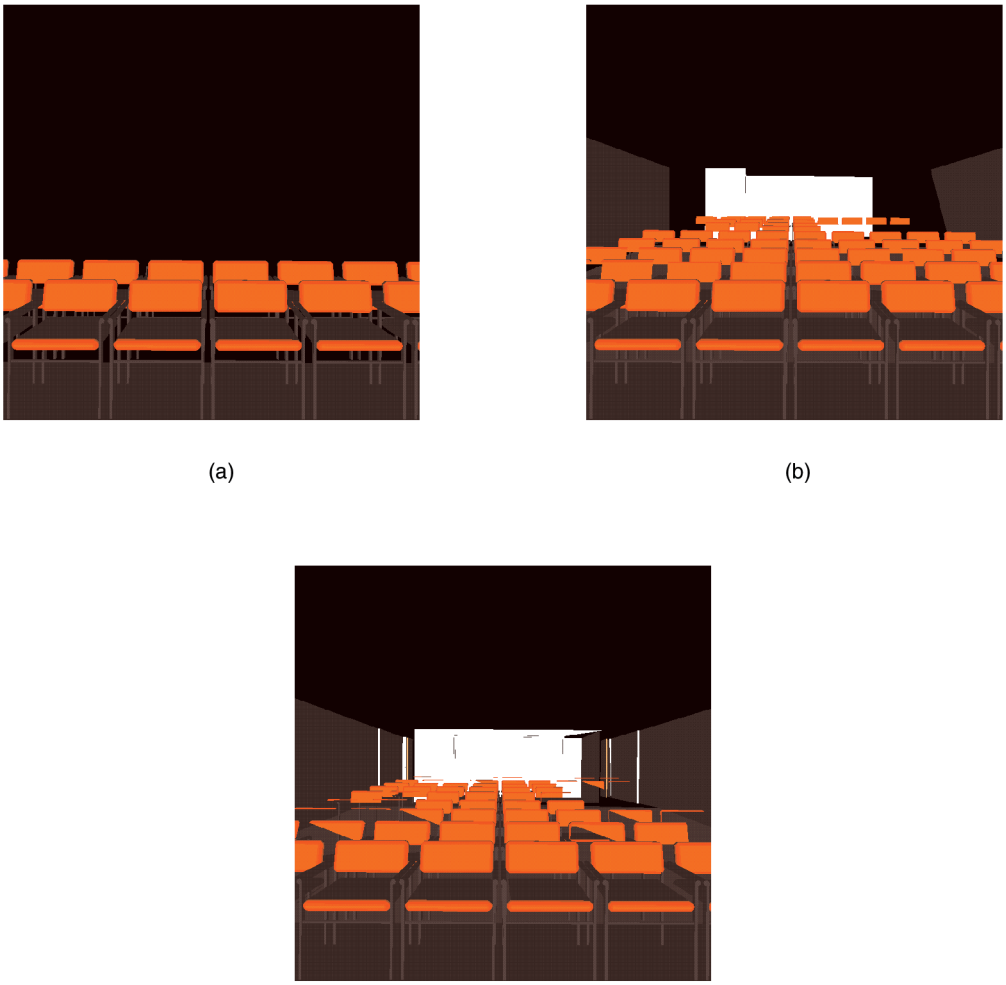


Fig. 13. This figure illustrates the qualitative differences among the different rendering techniques on each frame of the ROOM path. The three images show the actual rendered picture achieved with each rendering technique (centroid, octree-based PLP, and Delaunay-based PLP respectively).

octree approach was about four times faster. We have not included the timings for the centroid-sorting method as our implementation was straightforward and naively sorted all of the triangles for each of the frames. Fig. 12a highlights the effectiveness of our various methods for a budget of 10 percent of the total number of triangles, showing the number of visible triangles that were found. The magenta curve shows the exact number of visible triangles for each frame of this path. In comparison, the Delaunay triangulation was very successful, finding an average of over 90 percent of the visible triangles. The octree was not as good in this case and averaged only 64 percent. However, this was still considerably better than the centroid-sorting approach, which averaged only 30 percent. Fig. 12b highlights the effectiveness of the PLP approaches. In the worst case, the Delaunay triangulation version produced an image with 4 percent of the pixels incorrect with respect to the actual image. The octree version of PLP was a little less effective, generating images with at most 9 percent of the pixels incorrect. However, in comparison with the centroid-sorting method, which rendered images with between 7-40 percent of the pixels incorrect, PLP has done very well.

5.2.2 *Visual and Quantitative Quality on the ROOM Model*

Figs. 13 and 14 show one of the viewpoints for the path in the Seminar Room dataset. Most of the geometry is made up of the large number of chairs, with relatively few triangles being contributed by the walls and floor. From a viewpoint on the outside of this room, the walls would be very good occluders and would help make visibility culling much easier. However, once the viewpoint is in the interior sections of this room, all of these occluders are invalidated (except with respect to geometry outside of the room), and the problem becomes much more complicated. For a budget of 10 percent of the triangles, we provide figures to illustrate the effectiveness of our PLP approaches, as well as the centroid-sorting algorithm. Fig. 13 shows the images rendered by the centroid method, the octree method, and the Delaunay triangulation method, respectively. The image produced by the octree in this case is the best overall, while the centroid-sorting image clearly

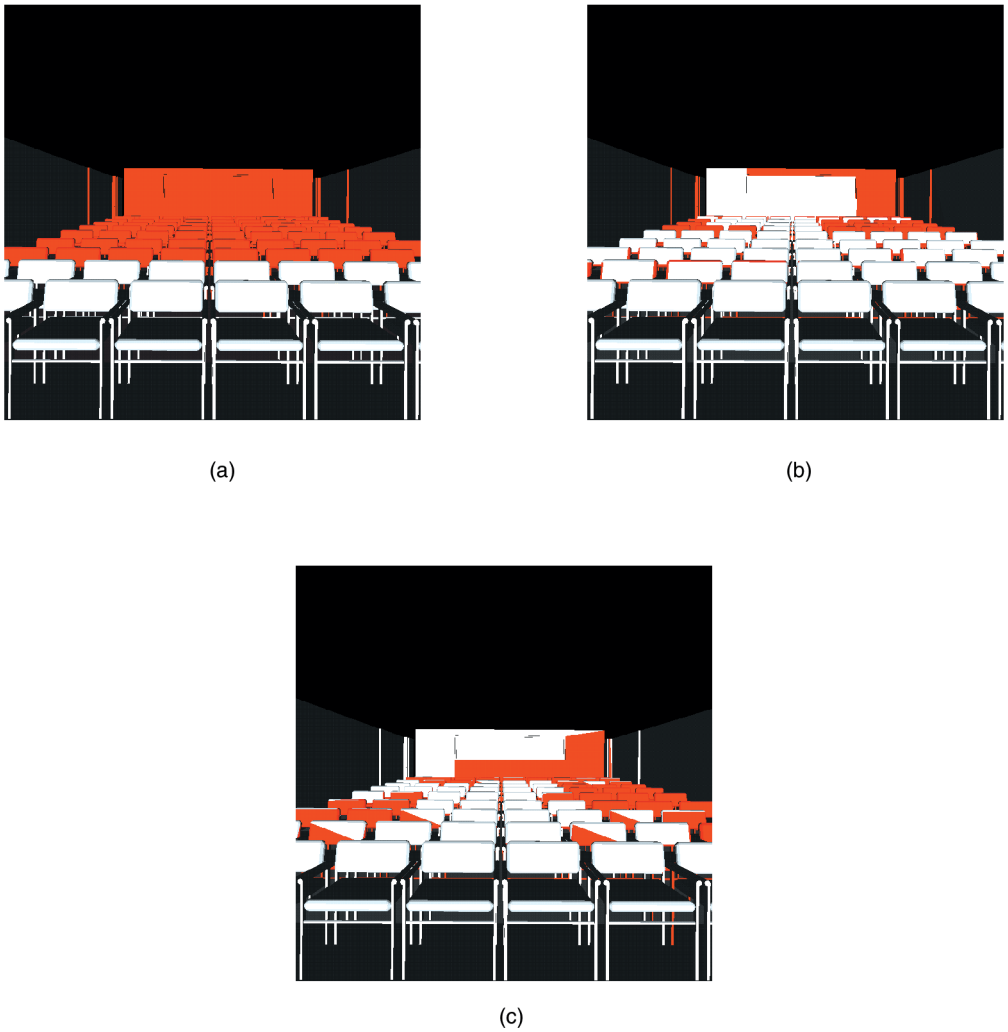


Fig. 14. This figure illustrates the qualitative differences among the different rendering techniques on a single frame of the ROOM. The three images show the missed polygons, rendered in red, to highlight which portion of the image a given technique rendered incorrectly.

demonstrates the drawback of using such an approach. To better illustrate where the algorithms are failing, Fig. 14 shows exactly the pixels which were drawn correctly, in white, and those drawn incorrectly, in red. Further quantitative information can be seen in Fig. 15. In fact, it is quite interesting that, in terms of the overall number of visible primitives, the centroid technique actually does quite well. On the other hand, it keeps rendering a large number of incorrect pixels.

5.2.3 *Summary of Results*

PLP seems to do quite a good job at finding visible triangles. In fact, looking at Figs. 10a and 11a, we see a remarkable resemblance between the shape of the curve plotting the exact visible set and 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 cannot be captured by a technique that relies on distance alone.

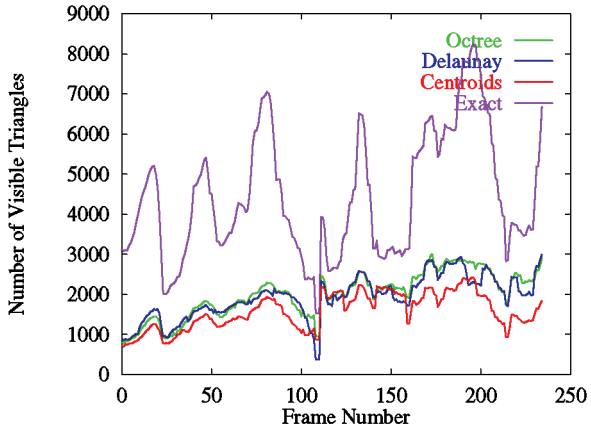
Still, we would like PLP to do a better job at approximating the visible set. For this, it is interesting to see where it fails. In Figs. 10d and 11d, we have 10 percent-budget images. Notice how PLP loses triangles in the back of the cars (in Fig. 11d) since it estimates them to be occluded.

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

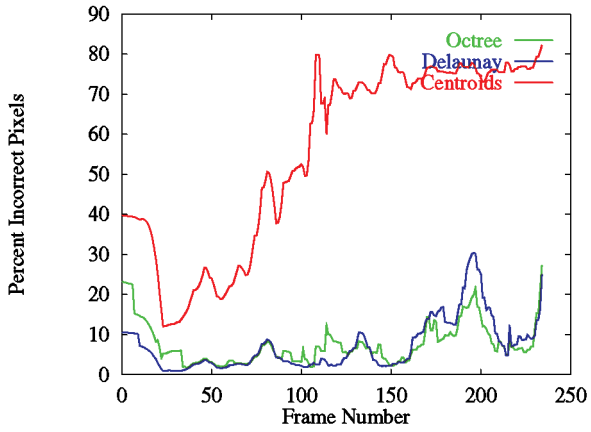
Overall our experiments have shown that: 1) PLP can be applied to large data, without requiring large amounts of preprocessing; 2) PLP is able to find a large amount of visible geometry with a very low budget; 3) PLP is useful in practice, making it easier to inspect large objects and in culling geometry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

6 Algorithm Extensions and Future Work

In this section, we mention some of the possible extensions of this work:



(a)



(b)

Fig. 15. This figure illustrates the quantitative differences among the different rendering techniques on a single frame of the ROOM. In each plot, we report results for each rendering technique (centroid, octree-based PLP, and Delaunay-based PLP, respectively). In (a), we show the percentage of the visible polygons that each technique was able to find. In (b), we show the number of incorrect pixels in the images computed with each technique.

1. Occlusion-culling techniques which rely on being able to use the z-buffer values to cull geometry, e.g., HOM [29], HP's occlusion-culling hardware [23], can potentially be sped up considerably with PLP.

   Take, for instance, the HP fx6 graphics accelerator. Severson [24] estimates that performing an occlusion-query with a bounding box of an object on the fx6 is equivalent to rendering about 190 25-pixel triangles. This indicates that a naive approach, where objects are constantly checked for being occluded, might actually hurt performance and not achieve the full potential of the graphics board. In fact, it is possible to slow down the fx6 considerably if one is unlucky enough to project the polygons in a back to front order (because none of the primitives would be occluded).

   Since PLP is able to determine a large number of the visible polygons at low cost in terms of projected triangles (e.g., PLP can find over 40 percent of the visible polygons while only projecting 1 percent of the original geometry). An obvious approach would be to use PLP's traversal for rendering a first "chunk" of geometry, then use the hardware to cull away unprojected geometry. Assuming PLP does its job, the z-buffer should be relatively complete, and a much larger percentage of the tests should lead to culling.

   A similar argument is valid for using PLP with HOM [29]. In this case, PLP can be used to replace the occluder selection piece of the algorithm, which is time consuming, and involves a nontrivial "occlusion preserving simplification" procedure.
2. Another potential use of the PLP technique is in level-of-detail (LOD) selection. The PLP traversal algorithm can estimate the proportion of a model that is currently visible, which would allow us to couple visibility with the LOD selection process, as opposed to relying only on screen-space coverage tests.
3. Related to 1 and 2, it would be interesting to explore techniques which automatically can adjust the PLP budget to the optimum amount to increase the quality of the images and, at the same time, decrease the rendering cost. Possibly, ideas from [12] could be adapted to our framework.

Besides the extensions cited above, we would like to better understand the relation of the solidity measure to the actual set of rendered polygons. Changing our solidity value computation could possibly 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 better the initial guess of what's visible, the easier it is to continue projecting visible polygons.

7 Conclusions

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.

As shown in this paper, PLP can be used with many different spatial tessellations, for example, octrees or Delaunay triangulations. In our experiments, we have found that the octree method is typically faster than the

Delaunay method due to its simple structure. However, it does not appear to perform as well as the Delaunay triangulation in terms of capturing our notion of polygon layering.

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, pregenerate occluders, and so on. Second, its flexibility to adapt to multiple machines with varying rendering capabilities. 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 percent of a scene and still be able to visualize over 80 percent of the visible polygons. If this is not accurate enough, it is simple to adjust the budget for the desired 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.

ACKNOWLEDGMENTS

The authors 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 Bernardini, Paul Borrel, João Comba, William Horn, and Gabriel Taubin for suggestions and help throughout the project. We thank Craig Wittenbrink for help with the occlusion-culling capabilities of the HP fx series accelerators, the Geometry Center of the University of Minnesota for `qhull` and `geomview`, and the University of California, Berkeley for the SODA Hall model used in some of our experiments.

REFERENCES

[1] D. Bartz, M. Meißner, and T. Hüttner, "Extending Graphics Hardware for Occlusion Queries in Opengl," *Proc. Workshop Graphics Hardware '98,* pp. 97-104, 1998.

[2] E.E. Catmull, "A Subdivision Algorithm for Computer Display of Curved Surfaces," PhD thesis, Dept. of Computer Science, Univ. of Utah, Dec. 1974.

[3] J.H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Comm. ACM,* vol. 19, no. 10, pp. 547-554, Oct. 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,* vol. 17, no. 3, pp. 243-253, 1998.

[5] J. Comba, J.T. Klosowski, N. Max, J.S.B. Mitchell, C.T. Silva, and P.L. Williams, "Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids," *Computer Graphics Forum,* vol. 18, no. 3, pp. 369-376, Sept. 1999.

[6] S. Coorg and S. Teller, "Real-Time Occlusion Culling for Models with Large Occluders," *Proc. 1997 Symp. Interactive 3D Graphics,* 1997.

[7] S. Coorg and S. Teller, "Temporally Coherent Conservative Visibility," *Proc. 12th Ann. ACM Symp. Computational Geometry,* pp. 78-87, 1996.

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

[9] D.P. Dobkin and S. Teller, "Computer Graphics," *Handbook of Discrete and Computational Geometry,* J.E. Goodman and J. O'Rourke, eds., chapter 42, pp. 779-796, Boca Raton, Fla.: CRC Press LLC, 1997.

[10] S.E. Dorward, "A Survey of Object-Space Hidden Surface Removal," *Int'l J. Computational Geometry Applications,* vol. 4, pp. 325-362, 1994.

[11] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics, Principles and Practice,* second ed. Reading, Mass.: Addison-Wesley, 1990.

[12] T.A. Funkhouser and C.H. Séquin, "Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments," *Computer Graphics (SIGGRAPH '93 Proc.),* J.T. Kajiya, ed., vol. 27, pp. 247-254, Aug. 1993.

[13] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," *Computer Graphics Proc., Ann. Conf. Series,* pp. 231-240, 1993.

[14] M. Held, J.T. Klosowski, and J.S.B. Mitchell, "Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs," *Proc. Seventh Canadian Conf. Computational Geometry,* pp. 205-210, 1995.

[15] P.M. Hubbard, "Approximating Polyhedra with Spheres for Time-Critical Collision Detection," *ACM Trans. Graphics,* vol. 15, no. 3, pp. 179-210, July 1996.

[16] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan, "Efficient Collision Detection Using Bounding Volume Hierarchies of k-dops," *IEEE Trans. Visualization and Computer Graphics,* vol. 4, no. 1, pp. 21-36, Jan.-Mar. 1998.

[17] M. Meißner, D. Bartz, T. Hüttner, G. Müller, and J. Einighammer, "Generation of Subdivision Hierarchies for Efficient Occlusion Culling of Large Polygonal Models," Technical Report WSI-99-13, ISSN 0946-3852, 1999.

[18] J.S.B. Mitchell, D.M. Mount, and S. Suri, "Query-Sensitive Ray Shooting," *Proc. 10th Ann. ACM Symp. Computational Geometry,* pp. 359-368, 1994.

[19] J.S.B. Mitchell, D.M. Mount, and S. Suri, "Query-Sensitive Ray Shooting," *Int'l J. Computational Geometry Applications,* vol. 7, no. 4, pp. 317-347, Aug. 1997,

[20] B. Paul and B. Bederson, "Togl—A Tk OpenGL Widget," http://Togl.sourceforge.net.

[21] J. Rohlf and J. Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," *Proc. SIGGRAPH '94, Computer Graphics Proc., Ann. Conf. Series,* A. Glassner, ed., pp. 381-395, July 1994.

[22] H. Samet, *Applications of Spatial Data Structures.* Reading, Mass.: Addison-Wesley, 1990.

[23] N. Scott, D. Olsen, and E. Gannet, "An Overview of the Visualize fx Graphics Accelerator Hardware," *The Hewlett-Packard J.,* pp. 28-34, May 1998.

[24] K. Severson, "VISUALIZE Workstation Graphics for Windows NT," HP product literature.

[25] C.T. Silva, J.S.B. Mitchell, and P. Williams, "An Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes," *Proc. ACM/IEEE Volume Visualization Symp. '98,* pp. 87-94, Nov. 1998.

[26] S.J. Teller and C.H. Séquin, "Visibility Preprocessing for Interactive Walkthroughs," *Computer Graphics (SIGGRAPH '91 Proc.),* T.W. Sederberg, ed., vol. 25, p. 61-69, July 1991.

[27] P.L. Williams, "Visibility Ordering Meshes Polyhedra," *ACM Trans. Graphics,* vol. 11, no. 2, 1992.

[28] H. Zhang, "Effective Occlusion Culling for the Interactive Display of Arbitrary Models," PhD thesis, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, 1998.

[29] H. Zhang, D. Manocha, T. Hudson, and K.E. Hoff III, "Visibility Culling Using Hierarchical Occlusion Maps," *SIGGRAPH 97 Conf. Proc., Ann. Conf. Series,* T.Whitted, ed., pp. 77-88, Aug. 1997.

James T. Klosowski received a BS in computer science and mathematics from Fairfield University in 1992, and an MS and a PhD in applied mathematics from the State University of New York at Stony Brook in 1994 and 1998, respectively. He is a research staff member at the IBM Thomas J. Watson Research Center. His main research interests are in computer graphics, visualization and applied computational geometry. While a graduate student, his research focused on applied computational geometry and computer graphics. He received the Catacosinos Fellowship for Excellence in Computer Science for his work in real-time collision detection. His research interests in computer graphics include interactive visualization of large datasets, collision detection, volume rendering, and adaptive network graphics. Recently, his research has focused on visibility culling and simplification of complex geometric models.

Cláudio Silva has a Bachelor's degree in mathematics from the Federal University of Ceara (Brazil), and MS and PhD degrees in computer science from the State University of New York at Stony Brook. He is a senior member of the technical staff in the Information Visualization Research Department at AT&T Labs-Research. His current research focuses on architectures and algorithms for building scalable displays, rendering techniques for large datasets, and algorithms for graphics hardware. Before joining AT&T, he was a research staff member at the graphics group at IBM T.J. Watson Research Center. There, he worked on 3D compression (as part of the MPEG-4 standardization committee), 3D scanning, visibility culling, and volume rendering. While a student and, later, as a U.S. National Science Foundation postdoctoral researcher, he worked at Sandia National Labs, where he developed large-scale scientific visualization algorithms and tools for handling massive datasets. His main research interests are in graphics, visualization, applied computational geometry, and high-performance computing. He has published more than 30 papers in international conferences and journals and presented courses at Eurographics and ACM Siggraph conferences. He serves on the committee for the IEEE Visualization 2000 Conference and the IEEE Volume Visualization and Graphics Symposium 2000. He is a member of the ACM, ACM Siggraph, IEEE, and IEEE Computer Society.