

An Extended abstract appeared in Proc. of Eurographics Workshop on Rendering, June 1996

# Hierarchical Back-Face Computation\*

Subodh Kumar      Dinesh Manocha      Bill Garrett      Ming Lin<sup>†</sup>

Department of Computer Science  
University of North Carolina  
Chapel Hill NC 27599  
USA

{kumar,manocha,garrett,lin}@cs.unc.edu

## Abstract

We present a *sub-linear* algorithm for computing and culling back-facing polygons that yields a significant performance improvement in the interactive rendering of large polygonal models. The algorithm partitions a polygonal model into hierarchical *clusters* based on the normals and positions of the polygons. It does not explicitly compute all the back-facing polygons but rather decides, in *expected constant time*, whether an entire cluster is back-facing. As a pre-processing step, the algorithm partitions the space into regions with respect to each cluster. During rendering, it exploits *frame-to-frame coherence* to track the view-point. The algorithm has been applied to a number of models and its performance is a function of number of clusters, the depth of the hierarchies, and the characteristics of the graphics system. In practice, we are able to cull 30 – 55% of the polygons in about 5 – 10% of the total CPU time per frame on an SGI Indigo2 Extreme for models composed of tens of thousands of polygons. It improves the overall frame rate by 30 – 70% as compared to hardware back-face culling.

**CR Categories and Subject Descriptors:** I.3.3 [Computer Graphics]: Picture/Image Generation — Display algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling, Rendering.

---

\*Supported in part by ARPA ISTO Order No. A410, NSF Grant No. MIP-9306208, Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, NSF Grant CCR-9319957, NSF Grant CCR-9625217, ONR Contract N00014-94-1-07 38, ARPA Contract DABT63-93-C-0048 and NSF/ARPA Center for Computer Graphics and Scientific Visualization. Approved by ARPA for Public Release - Distribution Unlimited

<sup>†</sup>Also with U.S. Army Research Office

# 1 Introduction

Given a polygonal model with well-defined normals, all the polygons whose normals point away from the view-point are called *back-facing*. If these polygons are part of solid polyhedra, they can be eliminated from the rendering process. This technique is known as *back-face culling* and is widely used to improve the frame rate. Many graphics systems implement it in hardware. On average, approximately one-half of the polygons of a polyhedron are back-facing. If the polygons are not part of a polyhedron or if the polyhedra have missing or clipped faces, back-facing polygons can be treated specially. If culling is not desired, the simplest approach is to flip the normal and treat the polygon as *front-facing* [6]. The simplest algorithms for back-face culling are based on computing the dot product of the polygon normal with a vector from the view-point to any point on the polygon. The complexity of these standard algorithms is linear in the number of polygons.

**Main Contribution :** We present a simple and sub-linear algorithm for hierarchical back-face culling. The algorithm has three main components:

**Cluster Formation:** The input polygonal model is partitioned into clusters based on the normals and physical proximity of the polygons. The cluster size is chosen as a function of the performance of the graphics system so as to optimize the overall performance. Large clusters are represented hierarchically.

**Spatial Partition:** The space is partitioned into *BackRegion*, *FrontRegion* and *MixedRegion* with respect to each cluster. These *Regions* are defined such that all the polygons in the cluster are back-facing or front-facing if the view-point is in the *BackRegion* or *FrontRegion*, respectively. Each *Region* is further decomposed into convex *Query* cells, each defined by three bounding planes.

**View-point Tracking:** At each frame the algorithm determines whether the view-point is in the *FrontRegion*, *BackRegion* or *MixedRegion* for each cluster. It starts with the *Query* cell(s) containing the view-point at the previous frame and incrementally computes the new *Query* cell(s) in *expected constant time*. If the view-point is in the *MixedRegion*, the algorithm is applied recursively to each child of the current cluster.

The space requirement of the algorithm is linear in the number of polygons, with a relatively small constant. Its overall performance is a function of number of clusters, the height of the trees and, and the motion of the view-point between successive frames. Performance is nearly independent of the number of polygons in a cluster. We have implemented and measured the algorithm's performance on different graphics platforms (SGI Indigo and Reality Engine systems) and on various models composed of tens of thousands of polygons. In practice, it improves the overall frame rate by 30 – 70% as compared to hardware back-face culling.

The rest of the paper is organized in the following manner. We survey the related work on visibility, polygon clustering, and use of coherence in Section 2. We introduce some terminology and give an overview of the algorithm in Section 3. In Section 4 we present the algorithm for computing the clusters and their hierarchical representation. Section 5 covers our spatial partitioning method. We describe the algorithms for tracking the view-point in Section 6 and its extension to dynamic environments in Section 7. We discuss implementation and performance on different platforms and models in Section 8.

## 2 Previous Work

There is a considerable volume of literature on hidden surface computation, polygon clusters, visibility computations, temporal coherence and on-line culling. In the early days of image synthesis a central geometric problem was visible surface computation. A number of algorithms have been proposed based on spatial partitioning, hierarchical representations, Z-buffer, list-priority, scan-line, area-subdivision and polygon clusters [6, 17]. It is still an active area of research in computational geometry, where many theoretically efficient algorithms have been proposed [3]. For models composed of tens of thousands of polygons, only Z-buffer approaches are able to give interactive performance on current graphics systems. Many non-interactive applications use binary space-partitioning (BSP) trees [7] to improve the rendering time of large static environments.

There is also significant amount of literature on the use of polygon clusters in visibility computations. Schumaker had earlier proposed them in [14] and Newell had used them as well [11]. The BSP tree algorithm is also based on the fact that environments can be viewed as being composed of clusters. Finally, coherence has been a key characterization of most visibility algorithms. In the classic paper, Sutherland et. al. had shown how visibility algorithms can take advantage of coherence and more than *eight* different kind of coherence were identified [17].

Rendering an extremely complex geometric database composed of millions of polygons has always been a challenge for visibility computations. To handle such large data-sets, three kind of visibility approaches have been used along with Z-buffer:

**View-Frustum Culling:** The technique of view-frustum culling uses spatial data structures like oct-trees and hierarchical traversals of such structures to cull out portions of the model not lying in the current *view volume* [4, 6].

**Obscuration Culling:** These techniques are used on scenes with high depth complexity and are based on hidden-surface removal methods and occlusion culling [6]. These include techniques based on partitioning the model into *cells* and *portals* and computing the partial visibility set (PVS) of polygons from each cell [1, 19, 20]. They have been successfully applied to architectural models and used to speed-up global visibility algorithms for illumination computation. A hierarchical Z-buffer algorithm combining spatial and temporal coherence with hierarchical structures has been presented in [9]. It cleverly exploits different types of coherence and can achieve several orders of magnitudes of acceleration compared with traditional techniques.

**Back-face Culling:** Back-face culling is a particular form of occlusion culling used on solid models that can be easily combined with other visibility culling methods. It has recently been extended to spline patches by computing a bound on the normals and Gauss map of a patch [10, 16]. The current implementations, however, take time linear in the number of polygons. Some techniques have been proposed to speed it up. Tanimoto [18] proposed a graph-theoretic approach that incrementally computes the silhouettes of a convex polytope using frame-to-frame coherence. It is difficult to extend it to general non-convex polytope models, though. The silhouettes of such models can have multiple components and the number of components vary from different view-point.

Other methods for back-face culling with sub-linear performance are based on *collating*

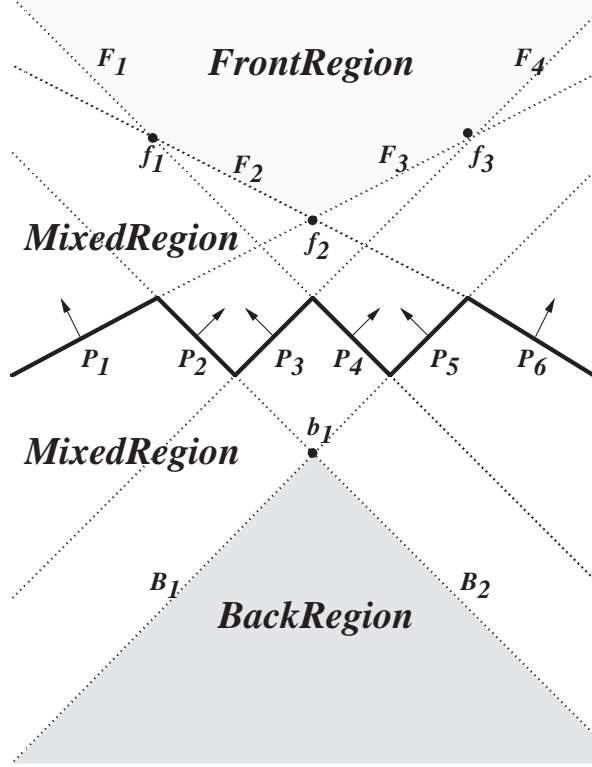


Figure 1: Different Regions of a Cluster

*global visibility* information, which can a priori compute which polygons are visible or back-facing from all the view-point. This is much more difficult than determining merely what is visible from a single view-point as is done in hidden surface removal. The fastest known algorithms currently used for computing a complete description of the interocclusion due to a polyhedral object with  $n$  vertices can take up to  $O(n^6 \log n)$  time [8].

### 3 Definitions and Algorithm Overview

For a polygon,  $P_i$ , in  $\mathcal{R}^3$ , let the equation of the plane containing  $P_i$  be  $a_i x + b_i y + c_i z = d_i$ . It partitions the  $\mathcal{R}^3$  into two half spaces:

$$\begin{aligned} \mathcal{H}_i^- &: a_i x + b_i y + c_i z < d_i \\ \mathcal{H}_i^+ &: a_i x + b_i y + c_i z \geq d_i \end{aligned}$$

Given the view-point,  $\mathbf{V} = (X, Y, Z)$ , polygon  $P_i$  is *back-facing* iff  $\mathbf{V} \in \mathcal{H}_i^-$  or *front-facing* iff  $\mathbf{V} \in \mathcal{H}_i^+$ . In the rest of the paper we use more relevant notation  $\mathcal{H}_i^b$  for  $\mathcal{H}_i^-$ , and  $\mathcal{H}_i^f$  for  $\mathcal{H}_i^+$ .

We generalize the concept of back-facing polygons for a cluster of polygons. For a cluster  $\mathcal{C}$  of polygons,  $P_1, P_2, \dots, P_m$ , we define:

**BackRegion**: the set of points in the intersection:

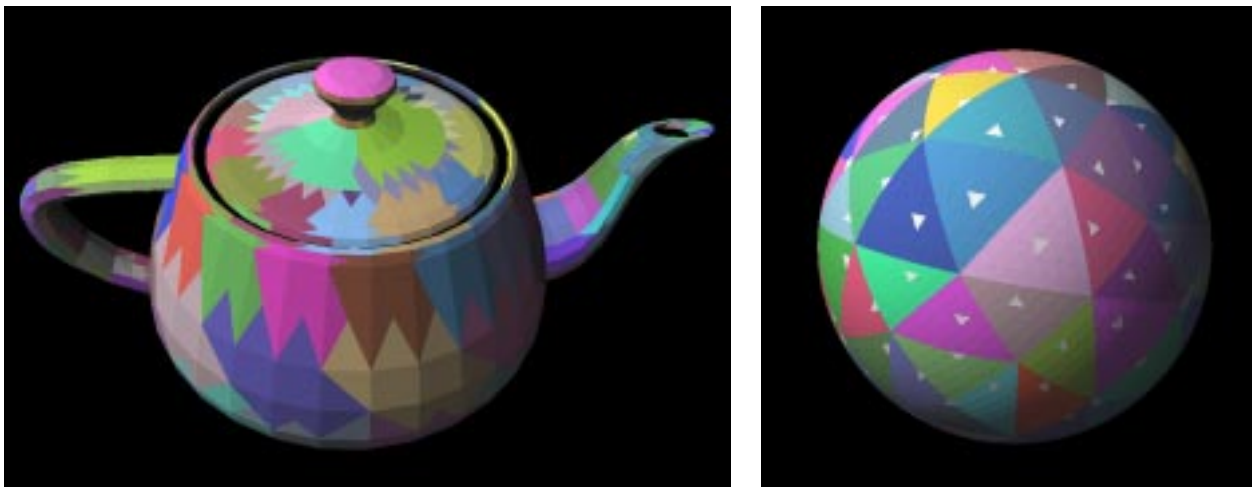
$$\mathcal{H}_i^b = \mathcal{H}_1^b \cap \mathcal{H}_2^b \cap \dots \mathcal{H}_m^b.$$

**FrontRegion**: the set of points in the intersection:

$$\mathcal{H}_i^f = \mathcal{H}_1^f \cap \mathcal{H}_2^f \cap \dots \mathcal{H}_m^f.$$

*MixedRegion*: the set of points which do not belong to either *BackRegion* or *FrontRegion*:  $\mathcal{R}^3 \setminus \text{BackRegion} \setminus \text{FrontRegion}$ , (where  $\setminus$  denotes set difference). Note that this spatial partition is specific to a polygon cluster. We illustrate these *Regions* for a collection of lines in 2-D in Fig. 1. Based on these definitions, it is clear that if the view-point  $\mathbf{V}$  lies in the *BackRegion*, then *all* the polygons in the cluster  $\mathcal{C}$  are back-facing. Similarly, if  $\mathbf{V}$  lies in the *FrontRegion*, all the polygons are front-facing. Finally, if the view-point lies in the *MixedRegion*, some of the polygons are back-facing and the others are front-facing.

Our algorithm proceeds by partitioning a polygonal model into clusters. Large clusters are organized hierarchically as a collection of (sub)clusters. The algorithm pre-computes the *Regions* for each cluster. Furthermore, each of these *Regions* is decomposed into convex *Query* cells, with three bounding planes. Given a *Query* cell, the algorithm can easily determine whether the view-point is contained in it. At run-time, our algorithm tracks the view-point with respect to the *Regions* of each cluster. For a cluster  $\mathcal{C}$ , if the view-point lies in its *BackRegion* (*FrontRegion*), respectively, all the polygons belonging to that cluster are back-facing (front-facing). Otherwise, some polygons in the cluster are front-facing while others are back-facing. In this case, the view-point is tracked with respect to the *Regions* of subclusters of  $\mathcal{C}$ . By combining hierarchical clustering, spatial partitioning and temporal coherence, we are able to rapidly cull away back-facing polygons and achieve significant improvement in visible surface determination for fast rendering.



(a) Teapot

(b) Tracked Sphere

Figure 2: Cluster formation

The overall algorithm makes use of a number of concepts and algorithms from computational geometry. We review them here. More details are given in [12].

**Convex Hull:** The convex hull of a set of points is the smallest convex set containing those points. A number of algorithms are known in the literature to compute the algorithms in 2-D and 3-D [12]. In our application, we use the Quickhull algorithm for computing convex hulls [2]. Its robust implementation is available as part of the Qhull public domain package.

**Linear Programming:** Geometrically, linear programming amounts to the following: given a set  $\mathcal{H}$  of half-spaces and a vector  $\mathbf{W}$ , compute a vertex  $\mathbf{v}$ , in the common intersection of

half-spaces, that *minimizes*  $\mathbf{v} \cdot \mathbf{W}$ . If this common intersection is null or unbounded, no such vertex exists. To solve a linear programming problem, we use the randomized algorithm presented in [15]. A public domain implementation of this algorithm is available.

**Duality:** Duality is powerful concept and used in a number of geometric algorithms [12]. In  $\mathcal{R}^3$ , the dual of a plane  $\mathcal{P} : ax + by + cz = 1$  is the point  $\overline{\mathcal{P}} : (a, b, c)$ . and vice-versa. (The dual of a line is a line.) Note that this normalized form of plane equation assumes that the plane does not pass through the origin. To handle this problem, the coordinate system is transformed, so that the new origin does not lie on the plane, i.e. we move the origin  $O'$  to a new point  $O$ . The dual is said to be taken about the point  $O$ . In our application, we are interested in the the following property of duality: The dual of the intersection of half spaces  $\{\mathcal{H}_i\}$ , is the convex hull of  $\{dual(\mathcal{H}_i)\}$ . Faces, edges and points on the dual hull correspond to points, edges and faces, respectively, on the boundary of the intersection of half spaces  $\{\mathcal{H}_i\}$ .

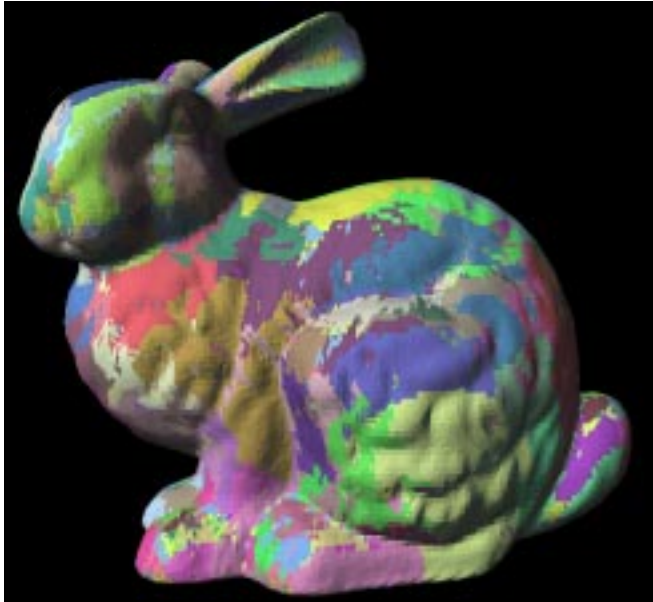
## 4 Cluster Formation

Given a model with  $N$  polygons, decomposing the model into clusters is an essential component of our algorithm. The choice of polygons in a particular cluster is governed by following constraints: How many cluster shall we partition the model into? Given the number of clusters, how shall we distribute the polygons amongst different clusters? We address the first question in Section 8.3 and present an algorithm to compute the number of clusters to maximize the overall performance. Given the number ( $Q$ ), we decompose the polygonal model into various clusters based on the following constraints:

- **Physical Proximity:** Minimize the physical proximity of the polygons. In other words, polygons in a cluster should not be far apart from each other, otherwise the view-point can be located between them.
- **Proximity in the Normal Space:** Maximize the  $\mathcal{B}ack\mathcal{R}egion$  of the cluster by reducing the variation of orientations of the polygons. This corresponds to minimizing their proximity in the normal space.

We use *duality* to minimize the proximity of the polygons in the normal space. Corresponding to each polygon in the primal space, we compute a point in the dual space. If the equation of the plane containing the polygon is  $a_i x + b_i y + c_i z = d_i$ , its corresponding coordinates in the dual space are  $(\frac{a_i}{d_i}, \frac{b_i}{d_i}, \frac{c_i}{d_i})$ . The proximity of the polygons in the normal space implies that their corresponding dual points are very close to each other in the dual space (based on the Euclidean distance between two points in the dual space). Thus, a simple algorithm for computing the clusters is based on computing the duals of all the polygons partitioning them into  $Q$  groups in the dual space such that all the points belonging to a group are “close” to each other. Such groups can be computed based on decomposing the dual space into grids, classifying the dual points with respect to grids and merging or splitting the grids as necessary (so as to obtain  $Q$  groups). The dual of the points in each group is a cluster in the primal space.

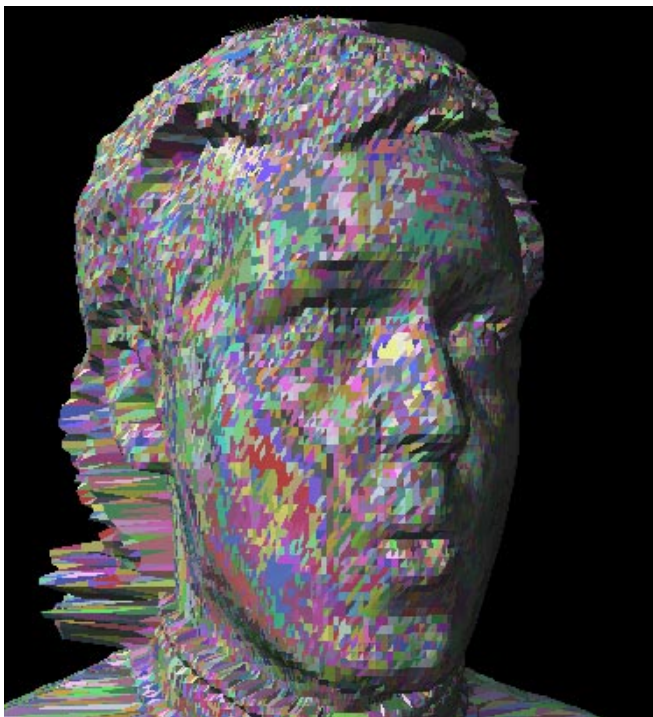
However, duality transformation and Euclidean distances in the dual space are not sufficient to satisfy both the constraints listed above. First, the dual points are not defined for



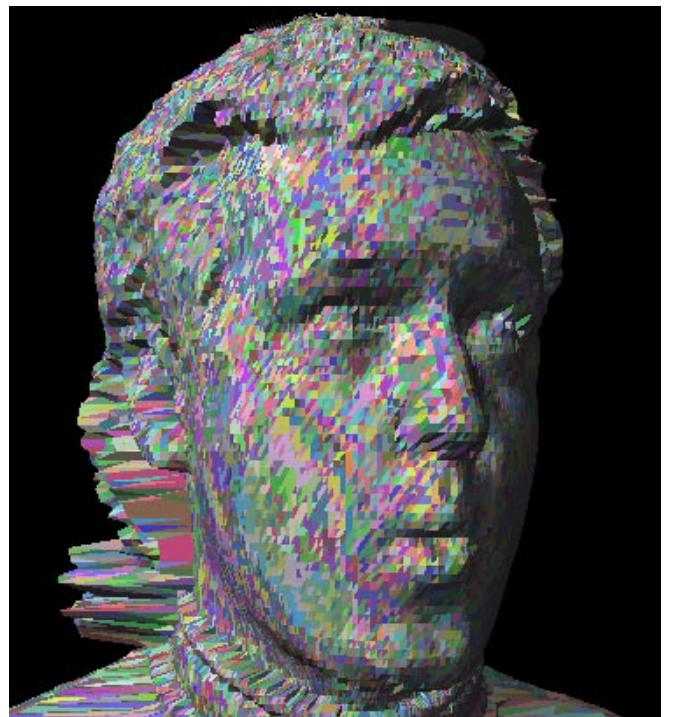
(a) Bunny (372 clusters)



(b) PLB (226 clusters)



(c) PLB (904 clusters)



(d) PLB (2536 clusters)

Figure 3: Clusters for the bunny and the PLB model

polygons, whose plane equation contains the origin. Second, polygons at distance  $l$  from the origin in the primal space get mapped to points at distance  $1/l$  in the dual space.

To circumvent these problems, we use a different metric in the dual space to compute distances between points. Given two points,  $\mathbf{A} = (\bar{x}_1, \bar{y}_1, \bar{z}_1)$  and  $\mathbf{B} = (\bar{x}_2, \bar{y}_2, \bar{z}_2)$ , the distance between them is defined as:

$$D(\mathbf{A}, \mathbf{B}) = \left| \frac{1}{\sqrt{x_1^2 + y_1^2 + z_1^2}} - \frac{1}{\sqrt{x_2^2 + y_2^2 + z_2^2}} \right|. \quad (1)$$

The resulting algorithm partitions the points in the dual space in different grids based on this metric. All the points in the same grid are merged into a cluster. Different grids are merged, so that we obtain  $Q$  clusters eventually. For the polygons whose plane equations pass through the origin, we associate them with the cluster which has the average normal closest to it (in terms of taking the dot products of the normalized vectors corresponding to the average normal and the polygon normal).

The resulting algorithm has been implemented and applied to various models. In Fig. 2, we demonstrate its performance to polygonal models of the teapot and sphere and in Fig. 3 to a bunny and the PLB model (recommended as a benchmark model by the Graphics Performance Committee). Each cluster corresponds to a different color on the model. In particular, it shows 48 clusters of the teapot in Fig. 2(a), 128 clusters of a sphere in Fig. 2, 372 clusters of the bunny in Fig. 3(a), and 226 clusters of the PLB model in Fig. 3(b).

## 4.1 Hierarchical Representation

Given a fixed number for clusters, the average number of polygons per cluster is high for large models. In such cases, the algorithm is not able to cull out most of the back-facing polygons. Increasing the number of clusters alleviates that problem, but the performance of the tracking algorithm is a linear function of the number of clusters. To circumvent these problems, we represent large clusters hierarchically.

Each cluster is decomposed into four or eight sub-clusters by subdividing the grids in the dual space based on the distance metric highlighted in (1). Each sub-cluster is decomposed recursively until the number of polygon is less than a user-specified threshold. In our implementations, we set the threshold to 20. Different hierarchical representations for a cluster are shown in Fig. 4. Given a cluster  $\mathcal{C}$ , its sub-clusters correspond to  $\mathcal{A}$  and  $\mathcal{B}$  in Fig. 4(b). The **BackRegion** of  $\mathcal{C}$  is  $R_C$ . The **BackRegion** of each sub-cluster is a *proper superset* of the **BackRegion** of the original cluster, as  $R_A = R_B \cap R_C$ . The sub-clusters corresponding to the first and second levels of the PLB model are shown in Fig. 3(c) and Fig. 3(d). They correspond to 904 and 2536 clusters, respectively.

## 5 Spatial Partition

Given a cluster,  $\mathcal{C}$ , of polygons,  $P_1, P_2, \dots, P_m$ . we present algorithms for computing the **Regions** of this cluster and decomposing them into **Query** cells.

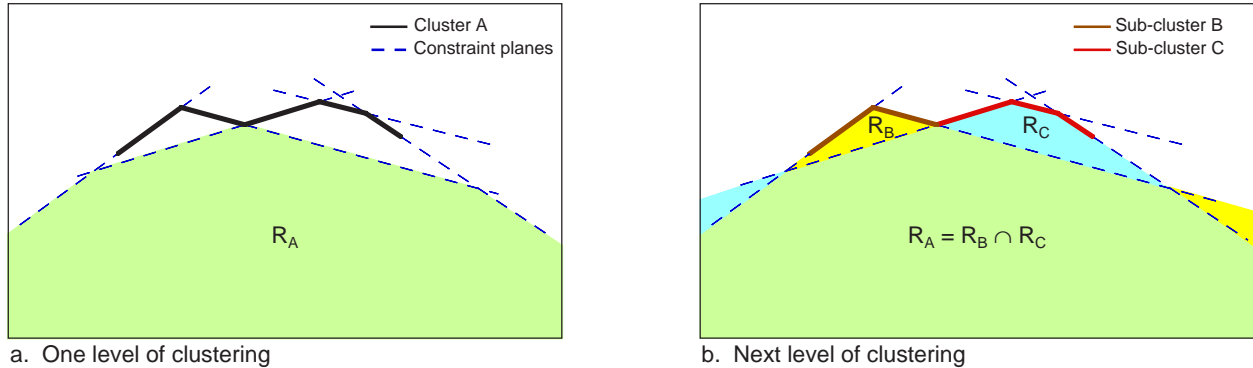


Figure 4: Hierarchical representation of Clusters

## 5.1 Regions Computation

The  $\mathcal{BackRegion}$  of a cluster is an open convex set consisting of boundary planes, and defined as the intersection of  $\mathcal{HB}_i$ 's. We need to compute:

- the boundary set  $h^b$  of planes that lie on  $\mathcal{BackRegion}$ .
- $l^b$ , the lines of intersection between adjacent boundary planes.
- $p^b$ , the points of intersection between adjacent boundary planes.

These are computed from the dual convex hull of  $\mathcal{H}^b$ .

### Algorithm I

1. Compute a point  $O \in \mathcal{BackRegion}$  (see Algorithm II).
2. Compute the dual point of each plane about  $O$ . Find convex hull of these points.
3. The points on the hull correspond to  $h^b$ . Since  $\mathcal{BackRegion}$  is an open set, not all faces of the hull correspond to a point in  $p^b$ . Only the faces whose normals point away from origin are *valid*.  $l^b$  corresponds to the edges of the hull between valid faces.

Next, we show how to compute the point  $O$ .

## 5.2 Feasible Point Computation

To compute the point  $O$ , we use linear programming. We can choose any minimization vector  $\mathbf{W}$ , and use the resulting vertex  $\mathbf{v}$ . However, for any random choice of  $\mathbf{W}$ , linear programming may return an unbounded solution. We present a simple algorithm, which computes a bounded point  $O$  inside the feasible region:

### Algorithm II

1. Choose a small  $\delta > 0$  and modify each half-space  $\mathcal{H}_i^b$  to:  $\underline{\mathcal{H}}_i^b : a_i x + b_i y + c_i z < (d_i - \delta)$ . It follows that  $\underline{\mathcal{H}}_i^b \subset \mathcal{H}_i^b$ .

2. Choose any polygon  $P_j$  of the cluster with the plane equation:  $a_jx + b_jy + c_jz = d_j$ , and set  $\mathbf{W} = (-a_j, -b_j, -c_j)$ .
3. Set  $O = \mathbf{v}$ , the vertex returned by the linear programming routine.

Based on our construction, it is clear that  $O$  is a vertex in  $\mathcal{B}\text{ackRegion}$ , at least a distance of  $\delta$  away from its boundary.

### 5.3 $\mathcal{F}\text{rontRegion}$ and $\mathcal{M}\text{ixedRegion}$ Computation

The algorithm for  $\mathcal{F}\text{rontRegion}$  computation is essentially the same as that for  $\mathcal{B}\text{ackRegion}$ . The  $\mathcal{F}\text{rontRegion}$  corresponds to the intersection of  $\mathcal{H}_i^f$ 's as opposed to  $\mathcal{H}_i^b$ 's. We do not compute the  $\mathcal{M}\text{ixedRegion}$  explicitly. It is implicit in  $\mathcal{R}^3 \setminus \mathcal{B}\text{ackRegion} \setminus \mathcal{F}\text{rontRegion}$ .

### 5.4 Decomposing into Query Cells

Given an arbitrary location of the view-point  $\mathbf{V}$ , a simple algorithm for point location initially tests each bounding plane for containment in the  $\mathcal{B}\text{ackRegion}$ . If the result is negative, it checks if the point is in  $\mathcal{F}\text{rontRegion}$ . Both  $\mathcal{B}\text{ackRegion}$  and  $\mathcal{F}\text{rontRegion}$  for a cluster are represented by the planes on their respective boundaries. As a result, any point location query can take time proportional to the number of bounding planes, which is  $O(m)$ ,  $m$  being the number of polygons in the cluster. In computational geometry literature, algorithms with logarithmic asymptotic complexity are known for point location in convex sets [5]. However, their space requirements and constant factors are rather high.

In this section, we present a simple algorithm to decompose  $\mathcal{B}\text{ackRegion}$ ,  $\mathcal{F}\text{rontRegion}$  and  $\mathcal{M}\text{ixedRegion}$  into  $\mathcal{Q}\text{uery}$  cells. Each  $\mathcal{Q}\text{uery}$  cell is a convex regions bounded by *three* planes. As a result, a point-location query in such cells can be answered in constant time. The tracking algorithm presented in Section 6 uses these cells along with temporal coherence to check which of the three  $\mathcal{R}\text{egion}$  contains the view-point. We present the decomposition algorithm for  $\mathcal{B}\text{ackRegion}$ , the same formulation applies to  $\mathcal{F}\text{rontRegion}$ . Our algorithm computes a center-point in  $\mathcal{B}\text{ackRegion}$ . Planes passing through this center-point and the boundary lines of  $\mathcal{B}\text{ackRegion}$  partition it into disjoint cells. It is easier to understand this partitioning in 2-D. Fig. 5 shows one such partitioning. The lightly shaded region shows the  $\mathcal{B}\text{ackRegion}$  of a cluster.  $\mathbf{CB}$  is the center-point. For each edge  $\mathbf{B}_j = \mathbf{b}_i\mathbf{b}_j$  on the boundary of  $\mathcal{B}\text{ackRegion}$ , the query cell  $\mathbf{BQ}_j$  is the open cone with  $\mathbf{CB}$  as its apex. These cells have the following property: if the view-point lies in  $\mathbf{BQ}_j$ , it is sufficient to test against the line  $B_j$  to determine if the view-point is in fact in  $\mathcal{B}\text{ackRegion}$ . Now we present the algorithm in 3-D:

#### Algorithm III

1. Compute the boundary of  $\mathcal{B}\text{ackRegion}$ , as outlined in Algorithm I.
2. Compute a center point  $\mathbf{CB} \in \mathcal{B}\text{ackRegion}$ . The arithmetic mean of boundary points is a good center point.

3. Each face on the boundary of *BackRegion* is defined by a sequence of edges. (This sequence is closed for polygons that have adjacent boundary planes on all their edges.) We triangulate the polygonal region, so that each face is bounded by three edges. Call the set of these triangular regions, **BT**.
4. For each **BT**<sub>*i*</sub> we construct three side-planes, each including an edge of **BT**<sub>*i*</sub>, and the point **CB**. Thus corresponding to each face **BT**<sub>*i*</sub> we get an open tetrahedral *Query* cell, **BQ**<sub>*i*</sub> with **CB** as its apex. If the view-point lies in a cell **BQ**<sub>*i*</sub>, the plane of face **BT**<sub>*i*</sub> determines containment in *BackRegion*.
5. Each edge on the boundary of *BackRegion* is shared by two facets. Hence each side-plane of a facet is shared by two *Query* cells. We use this adjacency information to maintain neighbor list for each *Query* cell. This is used to ‘walk’ to the appropriate cell when the view-point moves out of a given *Query* cell (as explained in section 6).
6. In addition to these cells, we add a truncated cone, that encloses the center-point. It is used to reduce the number of traversal steps (as discussed in section 6).

The algorithm similarly computes a decomposition of space into *Query* cells **FQ**<sub>*j*</sub><sup>'s</sup> based on the boundary of *FrontRegion*. It follows from construction that the two decompositions can overlap in *MixedRegion*. The algorithm keeps track of the overlaps amongst these query cells by storing the following information:

- We are given the boundary triangles of *BackRegion*, **BT**<sub>*i*</sub><sup>'s</sup>, and those of *FrontRegion* **FT**<sub>*i*</sub><sup>'s</sup>. For each **BT**<sub>*i*</sub> maintain pointers to all the *FrontRegionQuery* cells **FQ**<sub>*j*</sub><sup>'s</sup>, it overlaps. If there are more than three such *j*<sup>'s</sup>, subdivide **BT**<sub>*i*</sub> into smaller triangles such that the set associated with each sub-triangle has at most three cells.
- Repeat the process for **FT**<sub>*i*</sub> on the boundary of *FrontRegion*, storing their overlaps with **BQ**<sub>*j*</sub><sup>'s</sup>.

## 6 View-point Tracking

At run-time, the algorithm uses the pre-computed cluster descriptions to locate the view-point in the corresponding *Region* for each cluster. In particular, it keeps track of the *Query* cell(s) containing the view-point in the previous frame. If the view-point is contained in *BackRegion* or *FrontRegion*, it is a unique *Query* cell. On the other hand, if the view-point lies in *MixedRegion*, the algorithm stores a pair of *Query* cells of the form (**BQ**<sub>*i*</sub>, **FQ**<sub>*j*</sub>). At the current frame, the algorithm tests whether the view-point lies in the same cell(s). Each test involves testing the position of the view-point with respect to each bounding plane. If the view-point fails a bounding plane test, the algorithm *walks* to the neighboring cell adjoining that plane, as shown in Fig. 6 (red cell is the tracked cell). This walk is repeated until the algorithm finds the cell containing the view-point. If the resulting cell lies in the *MixedRegion*, the algorithm is applied recursively to each child of that cluster. Some extra processing is involved whenever the algorithm moves from *BackRegion* or *FrontRegion* to *MixedRegion* (or vice-versa). The running time of the algorithm is

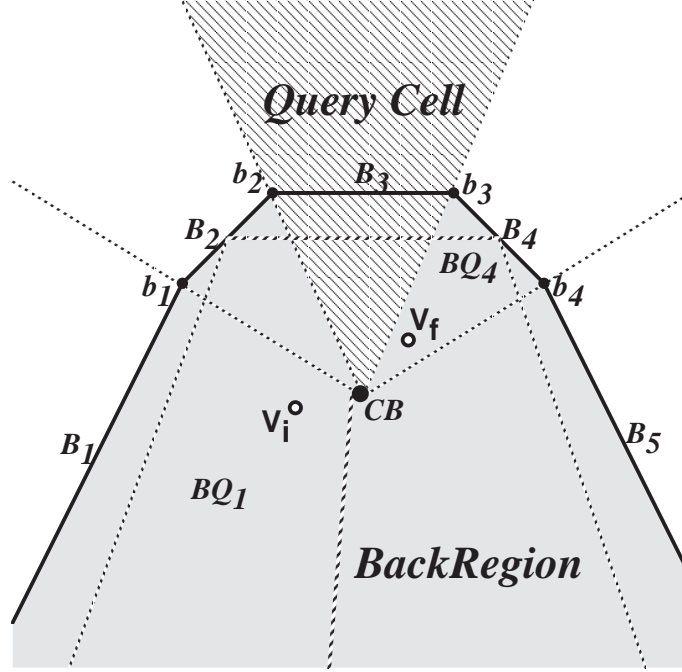


Figure 5: Tracking inside the Query Cells

a function of the number of cells traversed at that frame. Due to coherence, this number is typically a *small constant*.

In the first frame, the algorithm starts with a cell at random. Even in this case the expected number of cells traversed in a cluster with  $m$  equi-sized cells, is  $O(\sqrt{m})$ , if the cells uniformly partition the space. This is because the average length of the shortest path from one cell to the other is  $O(\sqrt{m})$ .

## 6.1 Reducing Traversals

Given that the performance for each cluster at every frame is determined by number of cells traversed, the algorithm minimizes the number of traversals. When the view-point fails the containment test for the current *Query* cell, it walks to a neighboring cell. If the view-point motion is small, for most cases the algorithm traverses only a few cells. But consider the case, when the view-point is very close to the center point **CB** for the *BackRegion* (as computed in Algorithm III), shown in Fig. 5. The initial position of the view-point is shown as  $\mathbf{V}_i$  in the cell  $\mathbf{BQ}_1$  (corresponding to the previous frame) and the final position is  $\mathbf{V}_f$  (corresponding to the current frame) in the cell  $\mathbf{BQ}_4$ . The traversal algorithm can visit  $O(m)$  cells for a cluster with  $m$  *Query* cells for a small motion. To circumvent such problems, the algorithm constructs a maximal inscribed cone inside the *BackRegion*, around the center-point **CB** (as shown in Fig. 5). (A similar inscribed cone is constructed for the *FrontRegion*.)

If the view-point was in *BackRegion* in the previous frame, the algorithm first tests whether the new location of the view-point is inside the inscribed cone of *BackRegion*. Only if this test fails, does the algorithm start traversing the *Query* cells. As a result, whenever the view-point is close to **CB**, the algorithm can trivially decide it is contained

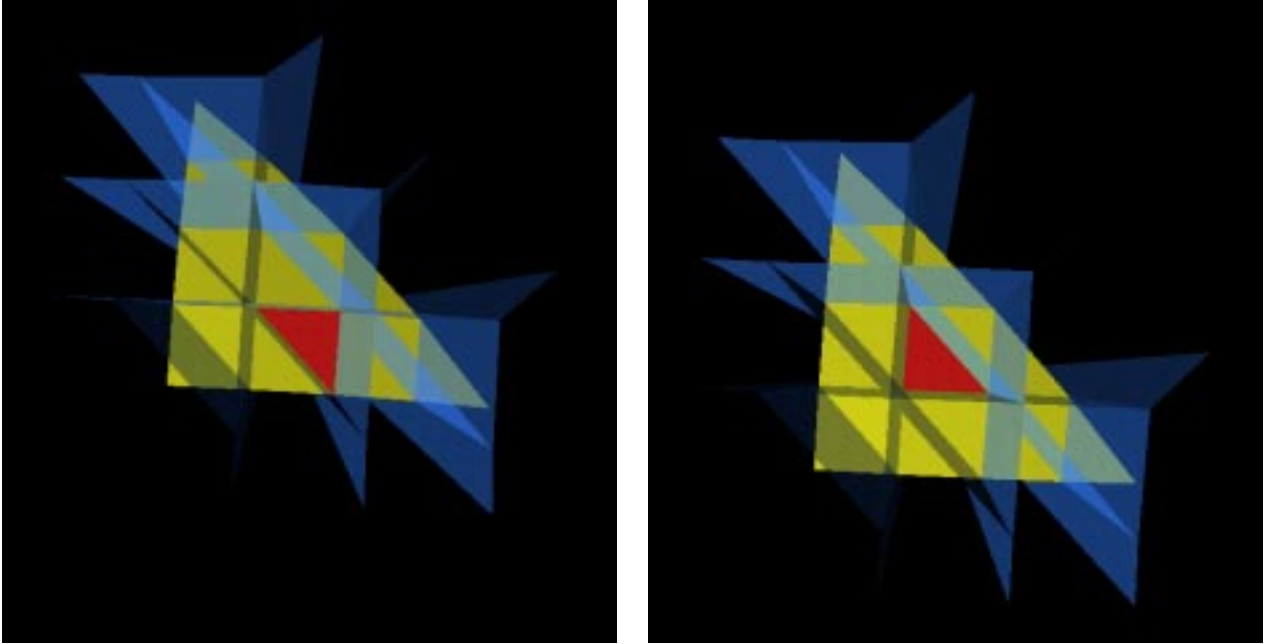


Figure 6: View-point Tracking

inside the *BackRegion*. A similar procedure is used if the view-point was in *FrontRegion* at the previous frame.

## 6.2 Traversing In and Out of *MixedRegion*

When the view-point moves into *MixedRegion* (from *BackRegion* or *FrontRegion*), the algorithm keeps track of two *Query* cells containing the view-point (of the  $BQ_i$  and  $FQ_j$ ). Assume that the view-point was in *BackRegion* at the previous frame and moves to the *MixedRegion* in the current frame. The traversal algorithm walks to a cell  $BQ_i$ . Let the bounding polygon of  $BQ_i$  correspond to  $BT_i$ .  $BT_i$  has associated with it, the  $FQ$ 's it overlaps. The algorithm starts with one of those cells, and tracks the view-point down to the appropriate  $FQ_j$ . The algorithm similar when the view-point moves from *FrontRegion* to *MixedRegion*.

When the view-point moves into *BackRegion* (*FrontRegion*) from *MixedRegion*, the algorithm stops tracking the query cell from *FrontRegion* (*BackRegion*).

## 7 Extension to Dynamic Environments

The algorithms for partitioning space and tracking the view-point described in the previous sections assume that the models are static and that only the view-point is changing between frames. In dynamic environments, however, some of the models may undergo motion (rotation or translation). Recomputing the clusters and space partitions is relatively expensive for interactive performance; instead, the culling algorithm is modified to take into account the motion transformations.

Whenever a polygon cluster undergoes a rigid transformation,  $T$ , the three **Regions** and the associated **Query** cells undergo the same transformation. The algorithm stores  $T$  and applies the inverse transformation to the view-point before tracking it with respect to the cluster. This assumes, of course, that all the polygons in a cluster undergo the same transformation. If only a subset of polygons are transformed, this technique doesn't work and the algorithm must re-compute the **Regions** and the **Query** cells.

## 8 Implementation and Performance

We have implemented the algorithms presented in this paper and tested them on a number of models using different graphics systems. The memory requirements of the algorithm is linear in number of polygons. The routines corresponding to cluster formation and space partition have not been optimized for performance. The performance of the run-time tracking routine varies considerably with the representation of data structures and memory organization. Our current implementation is not optimized for memory use, and we expect to improve the overall performance by 10 – 15% by optimizing. We plan to release our code as a *public domain* library in the near future.

### 8.1 Geometric Robustness

We used public domain packages for convex hull computation and linear programming. These packages are reasonably robust and perform well for most inputs. They occasionally fail, however, because of precision problems and certain degenerate inputs. We apply a small random *perturbation* of the input set (the vertices of the polygons) to overcome such problems for convex hull computations. It works well in practice.

Moreover, the minima of the linear programming problem in Algorithm II (step 2), is not unique and corresponds to any point on the bounding plane  $a_jx + b_jy + c_jz = d_j$ . Because of floating point precision problems, the linear programming algorithm may fail to compute a minima. To solve this problem, we *perturb the minimization vector* in the direction opposite to the expected minima. Since we do not a priori know the direction, the pre-processing algorithm searches for the right directions by perturbing each of  $x, y$ , and  $z$ , in both positive and negative directions.

In addition, the value of  $\delta$  used in algorithm II is important. If it is too small, the point **O** would be close to the boundary of the **BackRegion**, potentially causing the Qhull algorithm to fail. On the other hand, a large  $\delta$  can cause the intersection set (of the modified half-space) to become an empty set. Our algorithm again searches for a right value, using a bisection scheme.

Another problem that arises due to floating point precision is cycling between **Query** cells. If the view-point is very close to the boundary of a cell, the point location algorithm may not be able to compute the orientation of a point with respect to a plane accurately. As a result, the algorithm may *walk* back to a cell it had already visited. This can result in an infinite cycle. We avoid such cycles, by introducing *visit-counters* for each cell and plane and update their values to correspond to the global frame number. A plane with its visit-counter

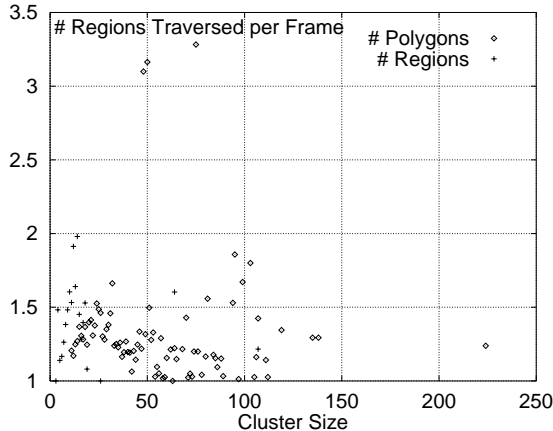


Figure 7: Number of *Query* Cells vs. Cluster Size

equal to the current frame number is not tested for point location again. In such cases, the algorithm assumes that the view-point is on the correct side.

## 8.2 Performance of the Tracking Algorithm

The running time of the tracking algorithm is primarily a function of the motion of the view-point between successive frames. It also depends inversely on the volume of the *Query* cells and sub-linearly on the size of the clusters. In Fig. 7, we highlight the average number of *Query* cells traversed at each frame as a function of the cluster size. The *X*-axis corresponds to the cluster size and the *Y*-axis is the average number of *Query* cells traversed. These data were generated on an SGI Indigo 2 Extreme (with a 250MHz R4400 processor) using different clusters of the PLB model (as demonstrated in the video).

In Fig. 8, we measure the performance of tracking algorithm as a function of the view-point motion (on a 250MHz R4400). These data was generated using the sphere model, in which each cluster has same number of polygons. The sphere is rotated about a fixed axis in space and the *X*-axis corresponds to the angle of rotation. The *Y*-axis is the average tracking time in microseconds. As the angle increases, the tracking time increases only sub-linearly. A very large value of rotation ( $> 120^\circ$ ) corresponds to the case of the view-point oscillating between *BackRegion* and *FrontRegion* in successive frames (from one end of the space to the other end); it corresponds to the *worst case* behavior for the tracking algorithm.

## 8.3 Number of Clusters

The performance of the culling algorithm is primarily determined by the number of clusters and is nearly independent of cluster size. Given an input model with  $N$  polygons, our goal is to divide it into  $Q$  clusters such that the overall performance is maximized. The choice of  $Q$  is governed by the following conflicting constraints:

- The running time of the culling algorithm increases linearly as a function of  $Q$ .

- The average number of polygons per cluster is  $N/Q$ . A small value of  $Q$  would imply that fewer polygons are culled or that the view-point is in the  $\mathcal{MixedRegion}$  of more clusters. In the latter case, the tracking algorithm is applied recursively to each sub-cluster, which increases the overall running time.

The optimum choice of  $Q$  is also a function of the graphics system. This includes the polygon rendering performance as well as the CPU performance. If polygon rendering is the bottleneck, the clustering algorithm can use a large value of  $Q$ . The extreme case is  $Q = N$ . In this case, the algorithm tests *each polygon* explicitly, to determine whether it is back-facing. This would also result in the maximum number of polygons being culled away. On the other hand, if the CPU performance is the bottleneck, the algorithm should use a low value of  $Q$ . The other extreme case is  $Q = 1$  and this implies no hierarchical back-face culling. In general, computing an optimum value of  $Q$  is non-trivial.

In our implementation, we have used the following heuristic to estimate a good value of  $Q$ . Given a graphics system, the algorithm can easily estimate the average tracking time per cluster per frame based on simple experiments (e.g.  $4us - 8us$  on a 250 MHz R4400). In many applications, only a small percentage of CPU time may be available for the culling algorithm. If the cluster size is  $Q$ , the algorithm typically tracks about  $2Q - 3Q$  clusters and sub-clusters at each frame (on an average). As a result, we try to maximize  $Q$  such that the total tracking time is not a bottleneck on the CPU. This heuristic works well in practice (as shown in Table 1)

## 8.4 Applications and Speed-Up

In Table 1, we have demonstrated the performance of our implementation on different models and compared it with hardware back-face culling. We used a SGI *Indigo<sup>2</sup> Extreme* (250 MHz R4400 with 128MB memory) as well as a SGI *RealityEngineII* (200 MHz R4400 with 512 MB memory). The polygon rendering performance of the latter is about five times better than the former. The graphics pipelines on these systems transform the polygons and check the normal of every transformed polygon to decide whether it is back-facing.

We computed the average percentage of polygons culled in each frame and the addi-

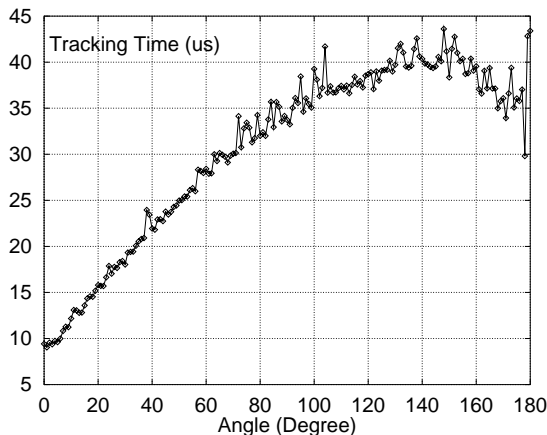


Figure 8: Tracking time as a function of “motion”

Model	Platform	Indigo2, Extreme Graphics			Onyx, Reality Engine		
	# Polygons	Polygons Culled	Frame-Rate Improvement	Tracking Overhead (CPU)	Polygons Culled	Frame-Rate Improvement	Tracking Overhead (CPU)
Sphere	2048	55.69%	41.91%	2.26%	55.69%	8.60%	6.63%
Bunny	69451	47.23%	71.05%	5.77%	47.23%	59.68%	11.97%
PLB	59079	37.54%	48.59%	6.29%	37.50%	35.15%	13.27%

Table 1: Performance Comparison

tional overhead on the CPU (by the hierarchical back-face culling algorithm) as well as the improvement in frame rate as compared to the hardware back-face culling algorithm. The performance varies with the graphics systems and the models. Typically, the algorithm is able to classify 75–85% of the model into *front-facing* and *back-facing polygons*. It does not render the back-facing polygons. Overall, it improves the frame-rate by 30–70% for large models. For relatively small models (like the sphere with 2,048 polygons), the algorithm does not produce much speed-up on high-end graphics systems. The additional overhead on the CPU of the tracking algorithm (as a percentage of the total frame time) is less than 10% on average. The algorithm performs extremely well on low-end graphics systems or on large-scaled complex models, whenever *polygon transformation is the bottleneck*.

## 9 Conclusion and Future Work

In this paper, we have presented a *simple* and elegant algorithm for hierarchical back-face computation. It is a *general purpose* algorithm applicable to all polygonal models and on *all graphics systems*. It can be easily integrated with all applications, whenever polygon rendering is the bottleneck. It is worth noting that the actual implementation can be further simplified by computing the regions in the dual space. If the back-face determination is also done in the dual space, we do not need to explicitly compute the boundary of intersection the half-space in primal space. The actual performance of the algorithm varies on different models and is also a function of the graphics systems. We have applied the algorithm to a number of models and are able to improve the frame rate by 30 – 70% in practice, with an additional overhead of up to 10% on the CPU. Future work include efficient implementation of parallel graphics systems as well as extension to dynamic models.

## 10 Acknowledgement

We thank Anselmo Lastra and Steve Molnar for insightful suggestions. Thanks to Greg Turk and Marc Levoy for the bunny model, and the graphics performance committee for the PLB head.

## References

- [1] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*,

- pages 41–50, 1990.
- [2] B. Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hull. Technical Report GCG53, The Geometry Center, MN, 1993.
  - [3] M. Bern, D. Dobkin, D. Eppstein, and R. Grossman. Visibility with a moving point of view. *Algorithmica*, 11:360–78, 1994.
  - [4] J.H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.
  - [5] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. In *Proc. 9th Internat. Colloq. Automata Lang. Program.*, volume 140 of *Lecture Notes in Computer Science*, pages 154–165. Springer-Verlag, 1982.
  - [6] J. Foley, A. Van Dam, J. Hughes, and S. Feiner. *Computer Graphics: Principles and Practice*. Addison Wesley, Reading, Mass., 1990.
  - [7] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *Proc. of ACM Siggraph*, volume 14, pages 124–133, 1980.
  - [8] Z. Gigus, J. Canny, and R. Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542–551, 1991.
  - [9] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proc. of ACM Siggraph*, pages 231–238, 1993.
  - [10] S. Kumar, D. Manocha, and A. Lastra. Interactive display of large scale nurbs models. In *Proc. of ACM Interactive 3D Graphics Conference*, pages 51–58, 1995.
  - [11] M. Newell, R. Newell, and T. Sancha. A new solution to the hidden surface problem. *Proc. ACM Ann. Conf.*, pages 443–448, 1972.
  - [12] F.P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.
  - [13] S. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *Proc. of ACM Siggraph*, pages 110–116, 1980.
  - [14] R. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual generation. Technical report, AFHRL-TR-69-74, US Air Force Human Resources Lab, 1969.
  - [15] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.
  - [16] L.A. Shirman and S.S. Abi-Ezzi. The cone of normals technique for fast processing of curved patches. In *EUROGRAPHICS*, pages 261–272, 1993.

- [17] I. Sutherland, R. Sproull, and R. Schumaker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.
- [18] S.L. Tanimoto. A graph-theoretic real-time visible surface editing technique. In *Proc. of ACM Siggraph*, pages 223–228, 1977.
- [19] S. Teller and P. Hanrahan. Global visibility algorithms for illumination computations. In *Proc. of ACM Siggraph*, pages 239–246, 1993.
- [20] S. J. Teller. *Visibility Computations in Densely Occluded Polyheral Environments*. PhD thesis, CS Division, UC Berkeley, 1992.