Partitioning Tree Image Representation and Generation from 3D Geometric Models

Bruce F. Naylor

AT&T Bell Laboratories
Murray Hill, NJ 07974
naylor@research.att.com

Abstract

While almost all research on image representation has assumed an underlying discrete space, the most common sources of images have the structure of the continuum. Although employing discrete space representations leads to simple algorithms, among its costs are quantization errors, significant verbosity and lack of structural information. A neglected alternative is the use of continuous space representations. In this paper we discuss one such representation and algorithms for its generation from views of 3D continuous space geometric models. For this we use binary space partitioning trees for representing both the model and the image. Our approach falls under the general rubric of visible surface algorithms, providing an object-space algorithm which under certain conditions requires only sub-linear time for a partitioning tree represented model, and in general exploits occlusion so that the computational cost converges toward the complexity of the image as the depth complexity increases. Visible edges can also be generated as a step following visible surface determination. However, an important contextual difference is that the resulting image trees are used in subsequent continuous space operations. These include affine transformations, set operations, and metric calculations, which can be used to provide image compositing, incremental image modification in a sequence of frames, and facilitating matching for computer vision/robotics. Image trees can also be used with the hemicube and light buffer illumination methods as a replacement for regular grids, thereby providing exact rather than approximate visibility.

Discrete vs. Continuous Space

We have come to think of images as synonymous with a 2D array of pixels. However, this is an artifact of the transducers we use to convert between the physical domain and the informational domain. Physical space at the resolution with which we are concerned is most effectively modeled mathematically as being continuous, that is, as having the structure of the Reals (or at least the Rationals) as opposed to the structure of the Integers. Modeling space as being defined on a regular lattice, while simple, is verbose and induces quantization which reduces accuracy and can introduce visible artifacts. Using nothing other than a lattice for the representation provides no image dependent structure such as edges.

Consider applying to a discrete image an affine transformation, an elementary spatial operation. The solution for this is developed by reasoning not merely in discrete space but in the continuous domain as well: samples are used to reconstruct a "virtual" continuous function which is then resampled. However, the quantization effects can become rather apparent should the transform entail a significant increase in size and a rotation by some small angle, despite the use of high quality filters. This is due to such factors as ringing, blurring, aliasing, and anisotropic effects which cannot all be simultaneously minimized (see, for example, [Mitchell and Netravali 88]). More importantly, discontinuities become increasingly smeared as one increases the size, since the convolution assumes a band-limited signal, i.e. an image with no edges. This has practical implications when texture mapping is used to define the color of surfaces in 3D: since a texture map can be enlarged arbitrarily, a brick texture, for example, will become diffuse instead of exhibiting distinctly separate bricks.

Now consider applying affine transformations to images represented by quadtrees, a spatial structure, developed within the context of a finite discrete space, for reducing verbosity and inducing structure on an image. The algorithm for constructing the new quadtree of the transformed image seems relatively complicated when compared to the corresponding algorithms for continuous space representations: it must resample each transformed leaf node and construct an entirely new tree. In contrast, boundary representations, simplical decompositions, or binary space partitioning trees only require transforming points and/or hyperplanes (a vector-matrix product), and

no structural changes are required. An extremal example of this difference is the quad-tree representation of a square occupying a quadrant, which requires 5 nodes, but when slightly rotated or translated the number of nodes is on the order of the number of pixels lying on its boundary (say about 4k for a 1k x 1k grid). This rather dramatic metamorphosis illustrates quite clearly that the quadtree reflects the nature of a finite discrete space, a nature differing from that of the continuum, and that applying arbitrary affine transformations in discrete space can affect the structure of the representation, introducing quantization noise and requiring more complicated algorithms.

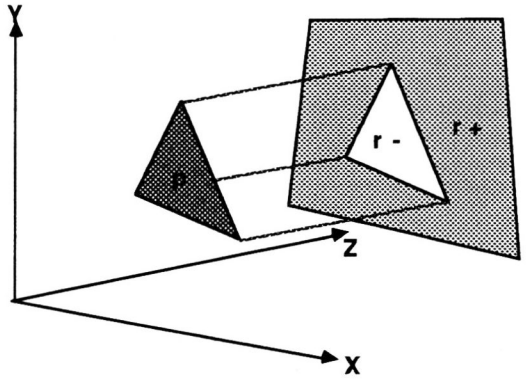
We are inclined to state a stronger proposition: discrete space, as a regular lattice, supports weakly the semantics of the continuum. Assuming this, the difficulties with transforming pixel arrays and quadtrees is not so unexpected. A good model for images is one that treats them as functions mapping a continuous 2D domain to a color space (the 2D domain may be unbounded). Discrete space representations are then treated as approximations of this function, or as evaluations achieved by point-sampling the domain, and discrete space operations are then constructed as approximations to their continuous space analogs. To display the image, conversion to a discrete representation would still be needed, but this now becomes strictly an issue of sampling the image function. (This argument should not be confused with the random vs. raster scan distinction, which is a question of transducer technology, not of computational technology.) With this said, we will now consider methods of generating continuous space image representations from 3D continuous space geometric models.

Visible Surface Algorithms

The context in which continuous space image representations are most easily produced is synthetic image generation. Here one begins with a 3D geometric model, defined using continuous space methods, from which a continuous space image representation is generated. This idea appeared very early in the development of visible surface algorithms, and in [Sutherland, Sproull and Schumacker 74] such algorithms were called *object-space* algorithms. But the approach has been neglected in favor of solutions utilizing quantized spaces (except in the Computational Geometry community).

The algorithm of [Weiler and Atherton 77] is a well known example of a continuous space method for generating images, and since it resembles closely our own approach, we will describe it in some detail. The algorithm operates on a set of polygons defined in a 3D post-perspective screen-space; thus, all projectors are parallel to the z-axis. Each polygon is represented by a boundary representation of some variety. Presumably the polygons are the faces of a collection of polyhedra, but this property is not relied on. The algorithm proceeds by recursively partitioning space until homogeneous regions of the image are generated. Homogeneity in this case means, in 2-space, a region in which only one polygon is visible, or in 3-space, a region which is entirely visible or entirely occluded. The output of the algorithm is a set of polygons in 2-space with disjoint interiors whose union forms the image. These polygons are in general non-convex and contain holes.

At each point in the recursion, a region **r** of space is partitioned into two sub-regions, which we denote as $\mathbf{r}^-$ and $\mathbf{r}^+$. The partitioning set used is a 3-space polygonal cylinder determined by the boundary of a polygon **p**, chosen from among those polygons that intersect **r** (Figure 1). The faces of the cylinder are orthogonal to the xy-plane, and so contain those projectors which go through the boundary of **p**. Since **p** may be of any genus, the sub-regions created by partitioning with **p** are not necessarily connected and are rarely convex. All polygons, including **p**, are then partitioned into subsets lying in $\mathbf{r}^-$ and $\mathbf{r}^+$, where we take $\mathbf{r}^-$ to be the sub-region containing **p**, i.e. the interior sub-region, and $\mathbf{r}^+$ to be the exterior sub-region.

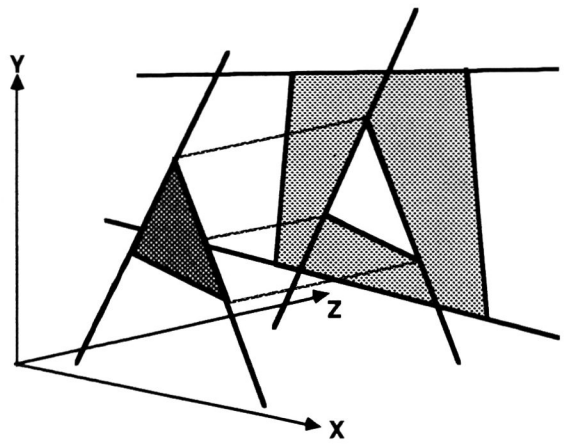


Weiler-Atherton Algorithm
Figure 1

Whenever there is some polygon $\mathbf{p}'$ in **r** with supporting hyperplane **h** such that $\mathbf{p}' = \mathbf{h} \cap \mathbf{r}$, then all polygons lying "behind" $\mathbf{p}'$ are occluded by $\mathbf{p}'$; such a polygon was called a surrounder in the literature on visible surface algorithms of the 70's, taken from the analogy of a 2D window being surrounded by a polygon. The algorithm selects whenever possible the plane of such a polygon as the partitioning set and then treats the "far region" of $\mathbf{p}'$ as homogeneous, i.e. as being totally occluded, and so terminates recursion in that region and discards the occluded polygons. Note that the cylindrical partitioning by **p** above results in **p** being a surrounder for $\mathbf{r}^-$. Finally, whenever a 3-space region is generated containing no polygons, this region is necessarily homogeneous.

While the algorithm as just described is all that is required to generate a set of polygons forming the image, we have said nothing about which polygon is chosen as the partitioner when no surrounder is present. A typical technique aimed at improving the performance is to initially sort the polygons in **z** using for each polygon the smallest z-coordinate from among its vertices, and then to maintain this ordering when polygons are partitioned. The partitioner in the absence of a surrounder is then the first in this ordering among those intersecting a region. In the presence of multiple surrounders, the closest one is chosen.

A similar but lesser known approach was described in an unpublished paper by Ivan Sutherland [Sutherland 73], where he develops a visible surface algorithm inspired by the ideas used in the 1D sorting algorithm *quicksort*. Its output is the same as the above algorithm, i.e. a set of disjoint polygons, and it differs from that algorithm primarily in one aspect: in the absence of a surrounder, the partitioning set is a plane through only one edge of a polygon in **r**. The plane then is orthogonal to the xy-plane, or equivalently, it contains the edge and the center of projection (Figure 2). Selecting which edge to use at each point in the recursion is a heuristic process. Sutherland tried several heuristics without reaching any firm conclusions about what method was best. It is interesting to note that Sutherland's paper also contains a section discussing how this algorithm can be used for shadow generation, transparency and collision detection.



Partitioning tree algorithm
Figure 2

As pointed out in [Harp 86], this algorithm can be treated as a binary space partitioning tree algorithm in that it uses a recursive partitioning by arbitrary hyperplanes; however, it does not generate a tree explicitly, a crucial distinction. This is not surprising given the original inspiration, quicksort. For indeed quicksort can be seen as implicitly constructing a 1D binary search tree, which in turn can be interpreted as a 1D partitioning tree. There is a somewhat subtle but important difference however: sorting has been developed in terms of points whereas space partitioning is in terms of hyperplanes. In 1D, and only in 1D, points and hyperplanes have the same dimension, viz. 0, and so it is easy to confuse them. But hyperplanes are (d-1)-dimensional not 0D sets, as are points. And they have an orientation that distinguishes the two halfspaces induced on a d-space, an orientation that can be used for ordering. Points have no such orientation, nor do they partition space, and so cannot be used to order d-space, d>1. This is one way to see why sorting algorithms are not applicable in dimensions other than 1D. Indeed, if we "attach" the ordering relationship to a 1D point, we then have a 1D hyperplane.

It seems apropos before leaving this section to discuss briefly the visible surface algorithm by John Warnock [Warnock 69]. It was the first recursive space partitioning, visible surface algorithm, and it follows the general scenario outlined above, the main difference being that the partitioning hyperplanes are not determined by polygonal edges (also, polygons were not explicitly partitioned). Today we see it as using a quadtree partitioning scheme. However, like the Sutherland algorithm, no explicit tree representation is generated; its output is a set of visible squares typically drawn directly into a pixel array. It is in effect a discrete space solution (also called a *screen-space* algorithm). It was to a certain degree the verbosity of this discrete solution that motivated the development of the two previously described continuous space algorithms.

Partitioning Trees

The binary space partitioning tree was originally developed in the context of visible surface determination. (The appendix contains a summary for those unfamiliar with the method.) [Schumacker et al 69] developed an incipient version that involved manual creation of a binary tree of vertical separating planes so that each object was separated from all other objects in the scene. The tree could then be used to generate a view-dependent visibility priority ordering. In [Fuchs, Kedem and Naylor 80] and [Naylor 81] three advancements were made: 1) the objects themselves were represented by the tree, 2) tree generation was automatic, 3) a dimension independent representation of space was introduced along with the name "binary space partitioning tree". As noted above, Sutherland also developed a number of ideas using this approach without generating a tree, the lack of which presumably contributed to his not realizing the connection between his work and that of [Schumacker et al 69].

A generalized view of partitioning trees sees them not simply as representations of polytopes but as a representation of functions whose domain and range are continuous spaces of finite dimensions d_1 and d_2 respectively: $f : X \in S^{d_1} \Rightarrow Y \in S^{d_2}$. The partitioning tree partitions the domain into a hierarchical collection of sub-domains. Within each sub-domain a value-continuous function f_i defines the value of **f** within that sub-domain (typically, f_i is defined for all of S^{d_1} as well, although this is not essential). All points in S^{d_1} at which **f** is value-discontinuous are contained within partitioning hyperplanes. This interpretation is relevant to the work here since images are functions from 2-space to some color space.

A partitioning tree also provides a structure enabling a hierarchical representation of **f**. As an example, consider polytopes. At the cells of the partitioning, each f_i is a boolean valued constant function indicating whether or not the cell is in the set. The polytope is the set of points **P = closure({ c_i | c_i is an in-cell})**. Thus **f** is the characteristic function f_χ for the set **P**; the algorithm for computing **f** is the point classification algorithm given in [Naylor 81] [Thibault and Naylor 87]. A useful hierarchical representation of f_χ can be obtained by associating with each region **r** a constant function providing the conditional probability of a point being in **P** given that it is known to lie somewhere within **r**. Thus, for example, the value at the root of the tree is the expected value of the function. We use this idea below to detect regions of the image plane that are discovered to be totally occluded yet inhomogeneous.

Partitioning Tree Visible Surface Algorithm

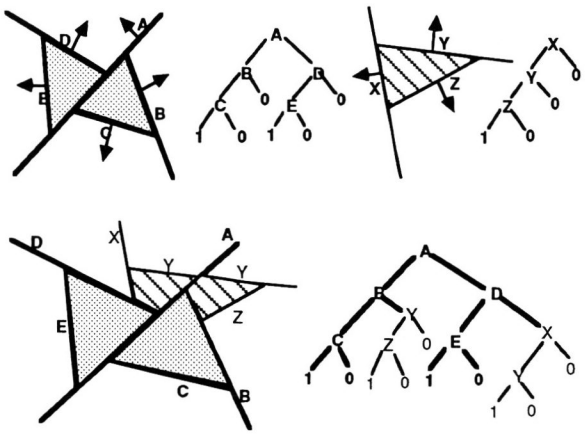
Consider a 3D geometric modeling system in which all geometric sets are represented by partitioning trees. An explicit representation of the model can be formed by taking the union of all the objects comprising the model, resulting in a single tree. This model-tree can then be used, along with a particular view of the model, to generate a total visibility priority ordering on the components of the tree. This ordering can be either far-to-near (back-to-front) or near-to-far (front-to-back). Generating this ordering can be combined with view volume clipping, which performs a non-destructive intersection operation between the model and the view volume, generally in sub-linear time [Naylor 90b].

Regardless of the ordering, a partitioning tree representing the image can be generated by forming the union of the faces in priority order. More specifically, consider a near-to-far ordering with the initial value of the image being the empty set.

1) project each face onto the 2D projection plane and let the attributes of each face be its color.

2) form the 2-space union of the faces in priority order:
   image = Union_Sets(image, face)
   where the attributes of the image take precedent over those of the face.

Thus faces are projected to 2D regions of the image plane, and the effect of higher priority faces occluding lower priority faces is achieved by letting the attributes of the image tree take precedence over those of the current face being "added" to the image. If the faces are represented by partitioning trees, then the union can be performed using tree merging [Naylor, Amanatides and Thibault 90] (figure 3), or if by b-reps, then by the algorithm given in [Thibault and Naylor 87]. If the reverse ordering is used (far-to-near), then the attributes of the new face would take precedence over those of the image tree. We see then that visible surface problem can be reduced to ordered set operations on polyhedral faces. (All of what has been said for 3D -> 2D holds for any d > 1, since partitioning trees and their algorithms are dimension independent.)



A union operation between two faces with attribute precedence
Figure 3

With either ordering, non-refractive transparency can be supported by using a "merge attributes" method that blends colors according to their opacity (alpha values). Given two polygons p_1 and p_2, in which p_1 has color c_1 and opacity α_1 and occludes p_2 which has color c_2 and opacity α_2, then the resulting color is $c_{1,2} = (c_1 * \alpha_1) + (c_2 * \alpha_2) * (1-\alpha_1)$ and opacity is $\alpha_{1,2} = \alpha_1 + (1-\alpha_1) * \alpha_2$ [Porter and Duff 84].

In addition the set of visible edges can be generated, if desired, by performing a closure operation which determines for each sub-hyperplane which subsets have a heterogeneous neighborhood [Naylor, Amanatides and Thibault

90]. So for example, in Figure 3 the subsets of hyperplane **A** that have homogeneous neighborhoods and so would not be on a discontinuity in the image are those that separate the two polka-dotted cells or two out-cells. This then provides a continuous space visible edge (hidden line) algorithm as an additional step after the visible surface algorithm.

A 3D variant is obtained by transforming the faces into 3D post-perspective screen-space. Then each face is considered to define a polygonal cylinder as in [Weiler and Atherton 77]. The union operation is now on 3D cylinders bounded on the near side by the plane of the face (see Figure 2). The faces could then be added to the image tree in any order, although near-to-far still has advantages as discussed below. Note that the 3D image tree that this produces represents in continuous space exactly the same function represented in discrete space by the standard {frame-buffer, z-buffer} structure.

The algorithm can also be performed in model-space, in which the cylinders are instead cones whose conical-vertex is the center of projection. This then becomes the algorithm present in [Chin and Feiner 89] which they applied to shadow generation, instead of image/visible-surface generation[1]. The resulting model-space image tree can then be transformed by the viewing transformation into screen-space. Working in model-space is preferable numerically, as it avoids the problems encountered as a consequence of the non-linear perspective projection which com-presses the depth at rate of z^{-2}. Note that this problem can be ameliorated somewhat by attempting to match the distribution created by the projection to the distribution of floating-point representable numbers. Uniformly distributed points in model-space become more compressed by the perspective projection the greater the depth. Floating-point representable numbers become more dense the closer the value is to 0. The standard mapping of the near plane to z=0 and the far plane to z=1 results in a mismatch: the greater the model-space depth the further the projected depth-value is from 0. This is trivially rectified by mapping the far plane to 0, and the near plane to -1 if in a left-handed system or to +1 if in a right-handed system.

Now let us compare our algorithm, using a near-to-far ordering, both to the Weiler and Atherton algorithm and to the Sutherland algorithm. They are of course all quite similar. They recursively partition space using at each stage a binary partitioning set (i.e. any (d-1)-set that partitions a d-region into two d-regions), and the partitioning is determined by planes containing either a polyhedral edge and the center of projection and/or by planes of faces. Aside from differences arising from the availability of a priority ordering (to be discussed below), the relationship of our algorithm to Sutherland's is simple: the order of "edge selection", i.e. partitioning hyperplane selection, is pre-determined by the priority ordering of the faces and the tree representing each face. And of course, we explicitly construct a tree to represent the output.

The primary difference between our method and that of Weiler and Atherton, once again other than the priority ordering, is the representation of polygons: their representation is a variety of b-reps while ours is partitioning trees. This difference manifests both in the algorithms for set operation (between faces), and the form of the output (a graph vs. a tree). It is our contention that the set operation algorithm for b-reps are more complicated, slower, and less numerically robust than the corresponding algorithm for partitioning trees. Some early indication of this is suggested by the fact that the original set operation algorithm given in [Weiler and Atherton 77], which is based on a kind of parity counting of intersections, fails to handle co-incident boundaries correctly. A correct but more involved solution was presented later in [Weiler 80] based on Euler operations.

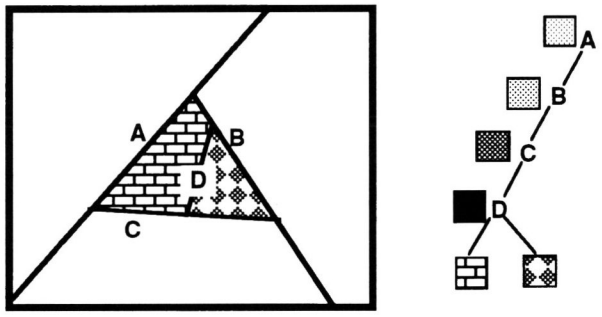
If in the case of partitioning trees, the boundary is already represented by 2D partitioning trees lying in a 3D hyperplane, as discussed in [Naylor 90a], then their representation in screen space either as 2-space entities or as 3-space cylinders is trivial, requiring the application of a single affine transformation (the inverse of the viewing transformation used for points). The tree merging algorithm can then be used to form the image tree as shown in Figure 1 above. Note that a single 2D partitioning tree can represent multiple coplanar connected components (faces), and a well built tree will generally yield better performance than operations on a list of connected components.

The second and more important difference arises from our use of a 3D partitioning tree to represent the model and so to generate a priority ordering. As a consequence, the algorithm is simplified by eliminating the code and execution time for the initial approximating depth sort, and everything associated with the notion of surrounders (detection and ordering). Of greater consequence is that the partitioning of faces by any

[1] Both their ideas and our ideas on this subject occurred independently. We first realized the potential presented here during the period in which we were developing the thesis that partitioning trees could provide a representation of polytopes [Thibault and Naylor 87]. Being able to solve analytically the visible-surface/shadow problems with partitioning trees, analogous to [Sutherland 73], provided part of the supporting evidence for this thesis. [Chin and Feiner 89] extended the ideas in [Thibault and Naylor 87] to generation of shadows. Concurrent with their work, we developed set operations on partitioning trees [Naylor, Amanatides and Thibault 90], which then enabled us to implement the work described in this paper. However, we had originally conceived of our solutions in terms of screen-space using 2D trees, rather than the model-space approach with 3D trees in [Chin and Feiner 89].

occluded subset of an edge is automatically eliminated simply by using a near-to-far ordering. Indeed, at any point in the construction, the image tree can be interpreted as a 2D-polytope representing a *visibility mask*: interior regions correspond to occluded regions and exterior to unoccluded regions. Since additions are made only to unoccluded regions, any intersection between two occluded edges is never computed.

Exploiting the creation of occluded regions of the image plane to reduce computation can be enhanced further by maintaining at internal regions a membership attribute indicating opacity within any region **r**. This can be either the percentage of **r** that is opaque, i.e. the expected value of a point lying in **r** being occluded (see Figure 4), or simply a boolean variable indicating whether **r** is fully occluded or not. Maintaining this membership value during the insertion of a new face amounts to the standard condensation of homogeneous regions used in set operations, the difference being that a region which is homogeneous only with respect to opacity but not color is not replaced by a leaf node. Thus, whenever an internal region becomes fully opaque, it will become a cell of the visibility mask even though a subtree remains defining the image within this fully occluded region. Consequently, this subtree is never again accessed during subsequent processing of lower priority faces. Moreover, when the root region becomes occluded, rendering ceases.



Maintaining % occluded at regions
Figure 4

This captures very simply the ideas present in other work using such masks[2] which require algorithms comparable to set operations on b-reps, and it is the continuous space correlate of pixel masks, be they 1-bit per pixel or many bits per pixel masks (i.e. sub-pixel masks) [Fiume and Fournier 83] [Carpenter 84]. When combined with view-volume clipping, an effect is achieved somewhat analogous to the culling methods presented in [Teller and Sequin 91]. While one would presume that their additional preprocessing would lead to noticeably less computation to generate an image, our method permits a dynamic geometric model (and of course none of the requisite preprocessing and storage of the resulting information).

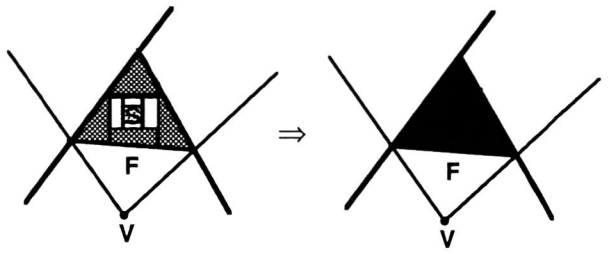
There is, however, a notable deficiency with our scheme as outline above: the order in which the image plane is partitioned is predetermined by the visibility ordering. However, the order in which hyperplanes are chosen affects significantly the "goodness" of the trees, i.e. the efficiency of the search structure provided by the tree. We have come to realize that an efficient partitioning tree is one that represents the set/function as something analogous to a sequence of approximations [Naylor 92]. We have implemented tree construction methods employing expected case models for various elementary operations and these methods produce such trees. What we would then like is to reflect within the image trees this effort at constructing good trees. To achieve this, instead of building the image tree from scratch, we modify (a copy of) the existing model tree so that it will become a representation in 3-space of the occluded and unocculded regions. This can be performed equally well in either model-space or screen-space, with the afore mentioned caveat that screen-space induces a numerically undesirable compression of the depth.

There are several ways to apply this idea; we will describe here only the simplest. We still traverse the tree in a near-to-far priority order. However, after forming the 3D face-beam, instead of performing **image** $\cup$ **face-beam** we perform **model** $\cup$ **face-beam**. As a consequence, entirely occluded faces will be removed by this union operation, and so will not have their face-beam constructed only to find that it is totally occluded, as will occur with the previous method. Indeed, every subtree of the model-tree that is found to be totally occluded will be condensed automatically by the union operation to a single cell before it is encountered in the priority traversal (Figure 5).

An extremal illustration of the power of this approach occurs when an entire object is occluded by a single face of another object. The beam for that face will "engulf" the object, and it will be reduced to a single "occluded" cell. Given our tree construction methods, the computation required in such a case is comparable to computing the union of the beam with a bounding volume of the object, yielding constant time elimination of the occluded object. Thus, under favorable conditions, the visible

[2] A recent example of this is [Sharir and Overmars 92], which is similar in many ways to our method, although it apparently was not implemented. They assume the existence of a visibility priority ordering, maintain a visibility map (our image tree) and a separate mask (our opacity attribute in the image tree). They also rely on merging of these. However, instead of adding one face at a time, they construct a separate visibility map for the next several faces, and then merge this with the "current" map, improving the worst case performance. This idea, if shown to be fruitful, can be easily applied to our method, since there is no algorithmic difference between a tree for a single face and another temporary image tree.

surface can be computed in sub-linear time (and this is in addition to the typically sub-linear clipping of the model to the view volume). More generally, this approach exploits during "beam insertion" the efficient search structures previously generated for each object and the gains from condensing homogeneous regions. Equally important, it retains a desirable residue of this structure in the resulting image tree, and this residue is important for efficient execution of any subsequent spatial operations, such as those discussed in the next section.



With viewer V, subtree S is occluded by face F and is removed by condensation.

Figure 5

Utilizing Image Trees

Given an image tree, one can sample it for display. There are a number of ways to do this. The simplest but most expensive would be to use point classification for each pixel to determine its color. This would, however, allow one to use non-uniform sampling techniques [Mitchell 87] for anti-aliasing. A more reasonable alternative would be to classify scan-lines. But since parametric representations are ideal for scan-conversion, and b-reps are in effect such representations, one can use the algorithm in [Thibault and Naylor 87] to classify an initial b-rep polygon corresponding to the viewport. This yields a disjoint set of convex polygons each as list of vertices, one for each cell, whose attributes are the color of the corresponding cell. Finally, if the faces of the polyhedra are given as b-reps, then these can be retained in the process that constructs the image tree, i.e. during the union operations, as described in [Naylor, Amanatides and Thibault 90]. Thus, by extracting these from the image tree, one obtains an output similar to that generated by the b-rep based algorithms, viz. a set of convex polygons each represented by a list of vertices.

Since only the visible surfaces are scan-converted, texture mapping and per-pixel illumination calculations (Phong shading) will be computed only for visible pixels. In addition, transparency is calculated between polygons rather than repeatedly for each pixel, and so can be provided on systems that do not have the requisite pixel-level hardware. The accuracy of anti-aliasing can be improved significantly, since the visible surface is represented at the resolution provided by floating point, which provides a much higher degree of accuracy than is practical with discrete space. Polygonal edges can be filtered using either continuous or discrete space representations of the filter, with the results being accumulated in the fame buffer using calculations analogous to those described for transparency. This then provides the continuous space version of sub-pixel mask techniques for anti-aliasing presented in [Fiume and Fournier 83] [Carpenter 84] and [Abram, Westover and Whitted 85], and no per-pixel list of micro-polygons with an approximating depth-sort is needed as in [Carpenter 84]. It is also a more efficient form of the per-pixel "analytic" approach in [Catmull 78] which relied on Sutherland's algorithm for visible surfaces. And for line drawings on B&W printers and displays, the visible edges can be used.

As discussed in the introduction, an immediate advantage of continuous space representations is that affine transformations can be applied with ease. Images can be scaled by (S_X, S_Y, S_Z) corresponding to a model-space scaling of $(S_X, S_Y, 1/S_Z)$. A rotation of an image about the screen-space z-axis by θ is comparable to a rotation by θ about the model-space image of this axis, which is the axis through the center of projection and orthogonal to the projection plane. A translation of the image by (T_X, T_Y) is equivalent to a shearing with respect to this same model-space axis by $(Sh_X = T_X, Sh_Y = T_Y)$. If the perspective is not too severe, then this approximates a similar translation in model-space. For defining texture on a surface, a 2D image tree can be affinely transformed in order to map it into screen-space and then sampled (note that transforming hyperplanes into screen-space requires no "perspective division", but only an affine transformation). In either case, no quantization artifacts, such as enlarged pixels or blurred edges, occur.

Image trees can also be used in subsequent continuous space operations. As noted above, this provides a continuous space version of what has been represented in discrete space by a rgbaz buffer. Therefore, compositing operations can be performed on 3D images as discussed in [Duff 85] (in that work, the space is discrete). These operations can be interpreted as set operations with blending. More specifically we have the following equality:

image(A <set op> B) =
image(A) <set op> image(B), <set op> $\in \{\cup, \cap\}$

And set difference can be used for masking. Since our 3D image trees are of the same data type as any other of our geometric sets, the previously developed set operations can give us compositing immediately. Blending is provided by the same mechanism that provides non-refractive transparency. Using 3D instead of 2D images frees the compositing from being simply a layering of images on top of each

other, as in the case for traditional cell animation or video games; i.e. visibility is not restricted to a total order on the individual images, instead they may be interleaved.

While compositing has not been associated with interactive 3D graphics, consider the rather likely situation in which a user has a model comprised of a collection of objects. Typically, the user will engage in modification of only one object at any given time while the view remains stationary. Then an image tree can be constructed for the static objects once at the beginning of this interaction, and the image of the model is generated by

image(model) = image(static-objects) ∪ image(dynamic-object)

This will yield more benefits the greater the number of static objects, the greater the amount of occlusion, and the greater the duration between selecting a new dynamic object. (For those readers familiar with random-scan display systems, each image tree is analogous to a segment.)

Additionally, it is possible to redraw into a frame-buffer only those faces whose visibility has changed between successive frames. To do this, one needs to maintain in the static-object's image tree a frame index at each node **v**. This will indicate the last frame in which the subtree rooted at **v** was changed by the union with the dynamic-object image tree. The drawing process needs to traverse only those subtrees which have changed in the current frame or else in the immediately prior frame so that the image of previously but no longer occluded static-object faces can be redrawn. This then provides a simple means of exploiting temporal correlation (frame-to-frame coherence) in this particular setting, i.e. static view and relatively few moving objects.

Visibility computations are, of course, crucial in the evaluation of all light transport equations. The equivalence between visible surface and shadow computations was recognized at a fairly early stage. Thus, our model-modification method can be used to partition model space into regions that are homogeneous in the number of lights visible from any point in that region, which then provides a way to classify any other set to determine its light-source visibility and simultaneously detect collisions. For global illumination, a well established technique is the hemicube method [Cohen and Greenberg 85] which for each surface element projects the scene onto a half-cube whose surface has been partitioned by a grid, and visibility is approximated within each grid-cell at the midpoint. Image trees provide an alternative to this grid. For each face of the hemicube, one can use our methods to represent the image. And instead of approximating the form factors discretely, the transport can be computed exactly using contour integration [Nishita and Nakamae 85] [Campbell 91]. This then leads to a global illumination algorithm with certain similarities to that of [Campbell 91] which is also based on partitioning trees. Similarly, image trees can be used to implement light buffers [Haines and Greenberg 86], once again, with exact rather than approximate visibility, yielding a significantly simplified methodology.

Generating image trees from 3D models provides a potentially important companion to our work on a discrete-to-continuous transform in which a pixel array representation is converted into a corresponding partitioning tree representation [Rahda et al 91]. Currently, we can solve no more than the segmentation problem; texture representation remains an open issue. However, this may be enough for certain applications. Consider a robotics application in which the constituents of an external environment are known *a priori* and for which a geometric model has been constructed. The problem is to maintain a correlation between an internal geometric model and the dynamic external physical state, given an initial correlated state. One could construct two image trees, one from the discrete image provided by a camera, and the other from the current view of the geometric model. These then could be correlated by an iterative process using affine transformations, set operations (symmetric difference) and calculation of moments, and in doing so determine how the geometric model should be updated. It may also be possible to use image trees in template matching.

Examples

Pictures 1-5 provide a few examples of generating image trees. The number of faces for pictures 1-3 are given below for each of the three rendering methods: painter's algorithm (method 1), creating a new image tree (method 2), and modifying the model tree (method 3). For the phone handset, the difference between method 3 and the number of front-facing polygons is due to the fact that the polygons forming the sound transmitting holes in the hand set are contained in fully occluded subtrees which are condensed to a single cell (the tree has also been clipped).

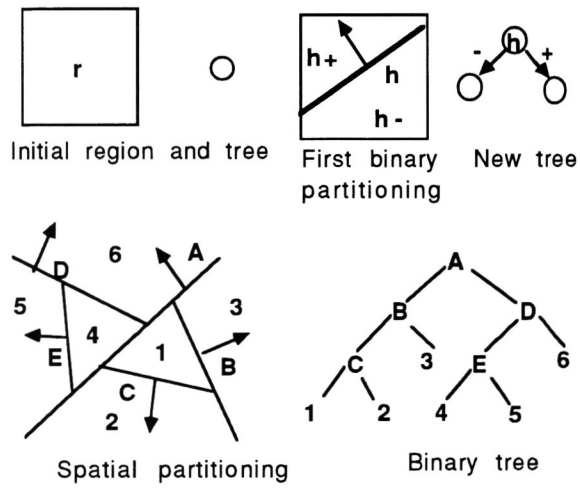
object	method 1	method 2	method 3
head	705	1982	703
shuttle	499	1100	523
phone	432	353	141

In Picture 4, we have composited two 3D image trees using a union operation. Picture 5 shows a skewed view of this revealing the solid nature of the images (the full boundary of the solid images has been generated only for the purpose of illustrating their 3D nature).

Appendix

Binary space partitioning trees, also called *bsp trees* or *partitioning trees*, are defined by a generating

algorithm, and for this only one operation is required: binary partitioning by a hyperplane of a region in a d-dimensional continuous space, d > 0. Figure A.1 illustrates this. Given a homogeneous open region **r**, a hyperplane **h** that intersects **r** is chosen using some criteria. Then **h** is used to induce a binary partitioning on **r** that generates two new d-dimensional regions, $\mathbf{r}^+ = \mathbf{r} \cap \mathbf{h}^+$ and $\mathbf{r}^- = \mathbf{r} \cap \mathbf{h}^-$, where $\mathbf{h}^+$ and $\mathbf{h}^-$ are the positive and negative *open* halfspaces of **h** respectively. Also, generated is a (d-1)-dimensional region $\mathbf{r}^0 = \mathbf{r} \cap \mathbf{h}$, called a *sub-hyperplane* (abbr. as *shp*). Thus $\mathbf{r} = \mathbf{r}^+ \cup \mathbf{r}^- \cup \mathbf{r}^0 = (\mathbf{r} \cap \mathbf{h}^+) \cup (\mathbf{r} \cap \mathbf{h}^-) \cup (\mathbf{r} \cap \mathbf{h})$. Any of these new unpartitioned homogeneous regions can be partitioned similarly, and so on recursively. When the process is terminated, the remaining unpartitioned regions, called *cells*, together with the sub-hyperplanes forms a partitioning of the initial region. (In figure A.1, the cells are labeled with numbers and the sub-hyperplanes with letters.)



Constructing a partitioning tree
Figure A.1

This process, when begun with d-space as the initial region, induces a structure on d-space in the form of a hierarchical decomposition. A partitioning tree is the computational representation of this process, and its combinatorial/syntactic form is captured by a binary tree. This tree is simply the directed graph of an asymmetric relation defined on the set of regions generated by the process where $\mathbf{r}_1 \rightarrow \mathbf{r}_2$ if $\mathbf{r}_2$ was created directly by a partitioning of $\mathbf{r}_1$. The tree also corresponds to the graph of the partial ordering of the regions induced by the subset relation. In addition, the tree can be interpreted as a type of computation graph by interpreting the arcs as intersection operations: "moving" a set **s** contained in a region **r** and partitioned by hyperplane **h** along a left arc from **r** to $\mathbf{r}^-$ can be interpreted as computing $\mathbf{s} \cap \mathbf{h}^-$, and similarly for the right arc. This interpretation provides a set theoretic definition of any region **r'** as the intersection of open halfspaces corresponding to arcs on the path from the root to **r'**. In figure A.1, **cell-3** = **2-space** $\cap$ $\mathbf{A}^-$ $\cap$ $\mathbf{B}^+$. Consequently, if the initial region is a convex and open set, it follows that all regions of the tree are convex and open.

A partitioning tree can provide the basis of a computational object for the semantic domain of *geometric sets*. These are subsets of continuous spaces of finite dimension for which each point has an associated set of attributes (e.g. color). The partitioning tree provides an isomorphism between certain geometric entities and a combinatorial structure manipulated by algorithms; in other words, the binary tree is a syntactic entity whose intended interpretation, or model (as in Model Theory), is a geometric set. In particular, a polytope, or collection of polytopes, can be represented by associating with each cell a *membership attribute* = { **in**, **out** }, dividing the cells into in-cells and out-cells. The polytope may be of any topology, including multiple connected components, and have a boundary that is non-manifold and/or unbounded. All possible trees represent some topologically valid polytope, although if a tree is chosen at random, certain subtrees may correspond to the empty set or to a homogeneous region. This means that every syntactically valid tree, i.e. any binary tree with hyperplanes at internal nodes and membership attributes at leaf nodes, represents a semantically valid polytope.

For any point in d-space, its ε-neighborhood with respect to the polytope can be discovered by following the paths in the tree to any cell whose closure contains the point. This is just the standard method of inserting a point into a search tree, with the simple extension that whenever a point is found to lie on a partitioning hyperplane, both subtrees are visited. The cells reached by the traversal are exactly those lying in the point's ε-neighborhood [Thibault and Naylor 87] .

Any central projection using linear projectors (rays) determines a partial ordering, called a *visibility priority ordering*, on the regions of any partitioning tree. This ordering depends only upon the center of projection. The total priority ordering induced by any single ray on the subset of the regions it intersects is consistent with this "global" partial ordering. The ordering is possible because for a ray **t** and hyperplane **h**, their intersection is a single point, unless **t** lies in **h**. This intersection point partitions **t** into near, on and far subsets. This implies that any set in the near-halfspace of **h** has priority over any sets lying in **h** which in turn has priority of any sets in the far-halfspace. Given a partitioning tree representation of polyhedra, discovering that the viewing position is in say the positive halfspace of a partitioning hyperplane **h** at node **v** means that all sets represented by the positive subtree of **v** have priority over any sets lying in **h** which then have priority over those sets represented by the negative subtree. One can apply this local ordering recursively to generate a total

priority ordering of all sets represented by the tree. (See [Schumacker et al 69] or [Sutherland, Sproull and Schumacker 74], and [Fuchs, Kedem and Naylor 80] or [Naylor 81]).

Along similar lines, efficient ray-tracing algorithms have been devised [Naylor and Thibault 86] which exploit both the convex decomposition and the inherent hierarchical search structure. Calculation of shadows due to point light sources is addressed in [Chin and Feiner 89] and due to area light sources in [Chin and Feiner 92] and [Campbell 91]. Use of partitioning trees for global illumination calculations can be found in [Fussell and Campbell 90] [Campbell 91]. Algorithms for set operations are presented in [Thibault and Naylor 87] and [Naylor, Amanatides and Thibault 90].

In [Naylor 81], it was shown that partitioning trees could represent arrangements of hyperplanes, and the complexity of the arrangements was used to give a bound of $\Theta(\mathbf{n^d})$ on the size of the largest possible partitioning tree formed using **n** hyperplanes in d-space. It was also shown that a set of disjoint **(d-1)-faces** could result in a tree of size $\Omega(\mathbf{n^{d-1}})$. In [Paterson and Yao 90] algorithms are given for converting a set of non-intersecting faces to a partitioning tree of size $\Theta(\mathbf{n^{d-1}})$ in time $\mathbf{O(n^{d+1})}$, $\mathbf{d > 3}$, which is reduced to $\Theta(\mathbf{n^2})$ and $\mathbf{O(n^3)}$ for 3D. In 2D, the tree size and run time are both $\mathbf{O(n \log n)}$. A convex n-gon can be represented by a tree of size $\Theta(\mathbf{n})$ and depth $\Theta(\mathbf{\log n})$ and two such trees can be merged in $\mathbf{O(n \log n)}$ [Naylor 92]. Two arbitrary trees each of size **n** can be merged in $\Theta(\mathbf{n^d})$ for $\mathbf{d} = 2, 3$ or 4 [Naylor, Thibault and Amanatides 90]. However, empirical results in [Naylor 81] and much subsequent experience indicate that polygonal models of real objects result in trees much closer to $\mathbf{O(n \log n)}$.

Partitioning trees are the same computational structure as *linear decision trees* [Rabin 72], which have been used to prove lower bounds on various problem, e.g. that sorting is $\Omega(\mathbf{n \log n})$. Another application of this structure, concerned primarily with representing a finite set of points is called *polygon trees* [Willard 82] or *partition trees.*

References

[Abram, Westover and Whitted 85]
Greg Abram, Lee Westover and Turner Whitted, "Efficient Alias-free Rendering using Bit-masks and Look-up Tables", **Computer Graphics** Vol. 19(3), pp. 53-59, (July 1985).

[Carpenter 84]
Loren Carpenter, "The A-buffer, an Antialiased Hidden Surface Method", **Computer Graphics** Vol. 18(3), pp. 103-108 (July 1984).

[Catmull 78]
Edwin Catmull, "A Hidden Surface Algorithm with Anti-Aliasing", **Computer Graphics** Vol. 12(3), pp. 6-10 (July 1978).

[Campbell and Fussell 90]
A.T. Campbell and Don Fussell, "Adaptive Mesh Generation for Global Diffuse Illumination", **Computer Graphics** Vol. 24(4), pp. 155-164, (August 1990).

[Campbell 91]
A.T. Campbell "Modeling Global Diffuse for Image Synthesis", Ph.D. Dissertation, Department of Computer Science, University of Texas at Austin, (1991).

[Chin and Feiner 89]
Norman Chin and Steve Feiner, "Near Real-Time Shadow Generation Using BSP Trees", **Computer Graphics** Vol. 23(3), pp. 99-106, (July 1989).

[Chin and Feiner 92]
Norman Chin and Steve Feiner, "Fast Object-Precision Shadow Generation for Area Light Sources Using BSP Trees", **Symp. on 3D Interactive Graphics**, (March 1992).

[Cohen and Greenberg 85]
Michael F. Cohen and Donald P. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments", **Computer Graphics** Vol. 19(3), pp. 31-40, (July 1985).

[Duff 85]
Tom Duff, "Compositing 3-D Rendered Images", **Computer Graphics** Vol. 19(3), pp. 41-44, (July 1985).

[Fiume and Fournier 83]
Eugene Fiume and Alain Fournier, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer", **Computer Graphics** Vol. 17(3), pp. 141-150, (July 1983).

[Fuchs, Kedem, and Naylor 80]
H. Fuchs, Z. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," **Computer Graphics** Vol. 14(3), pp. 124-133, (June 1980).

[Harp 86]
Keith Harp, "An Empirical Study of Visible Surface Algorithms" , Masters Thesis, School of Information and Computer Science, Georgia Institute of Technology, (Sept. 1986).

[Haines and Greenberg 86]
Eric A. Haines and Donald P. Greenberg, "The Light Buffer: a Shadow-Testing Accelerator", **IEEE Computer Graphics and Applications** Vol. 6(9), pp. 6-16, (Sept. 1986).

[Mitchell 87]
Don P. Mitchell, "Generating Antialiased Images at Low Sampling Densities" **Computer Graphics**, Vol. 21(4), pp. 65-72, (July 1987).

[Mitchell and Netravali 88]
Don P. Mitchell and Arun N. Netravali, "Reconstruction Filters in Computer Graphics" **Computer Graphics**, Vol. 22(4), pp. 221-228, (August 1988).

[Naylor 81]
Bruce F. Naylor, "A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes," Ph.D. Thesis, University of Texas at Dallas (May 1981).

[Naylor and Thibault 86]
Bruce F. Naylor and William C. Thibault, "Application of BSP Trees to Ray-Tracing and CSG Evaluation", Technical Report GIT-ICS 86/03, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332 (February 1986).

[Naylor 90a]
Bruce F. Naylor, "Binary Space Partitioning Trees as an Alternative Representation of Polytopes," **Computer Aided Design**, Vol. 22(4), (May 1990).

[Naylor 90b]
Bruce F. Naylor, "SCULPT: an Interactive Solid Modeling Tool," Proceeding of *Graphics Interface* (May 1990).

[Naylor, Amanatides and Thibault 90]
Bruce F. Naylor, John Amanatides and William C. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations", **Computer Graphics** Vol. 24(4), pp. 115-124, (August 1990).

[Naylor 92]
Bruce F. Naylor, "Constructing Good Partitioning Trees," manuscript in preparation.

[Nishita and Nakamae 85]
Tomoyuki Nishita and Eihachiro Nakamae, "Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection", **Computer Graphics** Vol. 19(3), pp. 23-30, (July 1985).

[Porter and Duff 84]
Thomas Porter and Tom Duff, "Compositing Digital Images", **Computer Graphics** Vol. 18(3), pp. 253-259, (July 1984).

[Rabin 72]
Michael O. Rabin, "Proving Simultaneous Positivity of Linear Forms", **Journal of Computer and Systems Science**, Vol. 6, pp. 639-650 (1972).

[Rahda et al 91]
Hayder Rahda, Riccardo Leonardi, Martin Vetterli and Bruce Naylor, "Binary Space Partitioning Tree Representation of Images", **Visual Communications and Image Representation**, Vol. 2(3), pp. 201-221, (Sept. 1991).

[Schumacker et al 69]
R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, "Study for Applying Computer-Generated Images to Visual Simulation," AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory (1969).

[Sharir and Overmars 92]
Micha Sharir and Mark H. Overmars, "A Simple Output-Sensitive Algorithm for Hidden Surface Removal," **ACM Transactions on Graphics** Vol. 11(1), (1992).

[Sutherland, Sproull and Schumacker 74]
I.E. Sutherland, R.F. Sproull and R. A. Schumacker, "A Characterization of Ten hidden Surface Algorithms," **A C M Computing Surveys** Vol. 6(1), (1974).

[Teller and Sequin 92]
Seth J. Teller and Carlo H. Sequin, "Visibility Preprocessing For Interactive Walkthroughs", **Computer Graphics** Vol. 25(4), pp. 61-69, (July 1991).

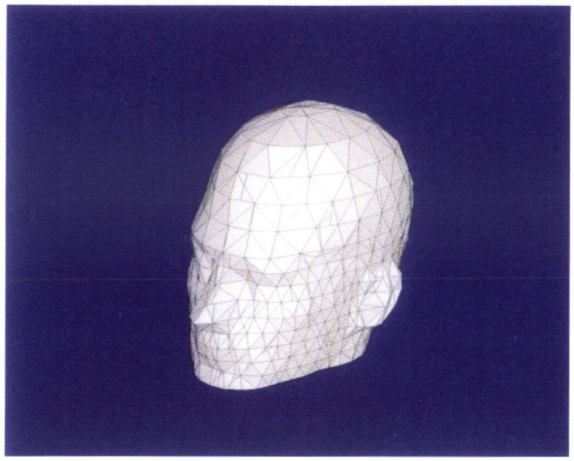
[Thibault and Naylor 87]
W. Thibault and B. Naylor, "Set Operations On Polyhedra Using Binary Space Partitioning Trees," **Computer Graphics** Vol. 21(4), pp. 153-162, (July 1987).

[Warnock 69]
John E. Warnock, "A Hidden-Surface Algorithm for Computer Generated Halftone Pictures", Computer Science Department, University of Utah, TR 4-15, (June 1969).

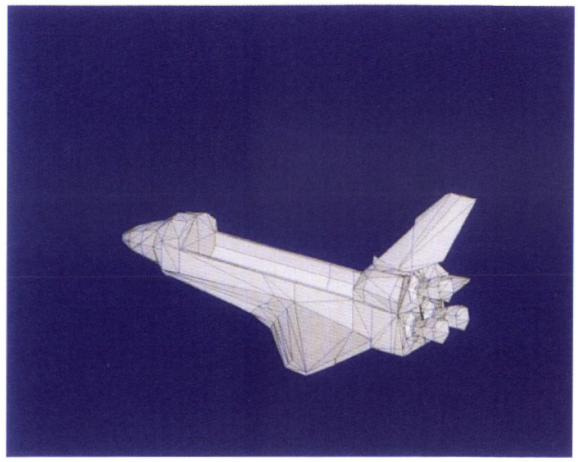
[Weiler 80]
Kevin Weiler, "Polygon Comparison Using a Graph Representation", **Computer Graphics** Vol. 14(3), pp. 10-18 (July 1980).

[Weiler and Atherton 77]
Kevin Weiler and P. Atherton, "Hidden Surface Removal Using Polygon Area Sorting", **Computer Graphics** Vol. 11(3), pp. 103-108 (July 1984).

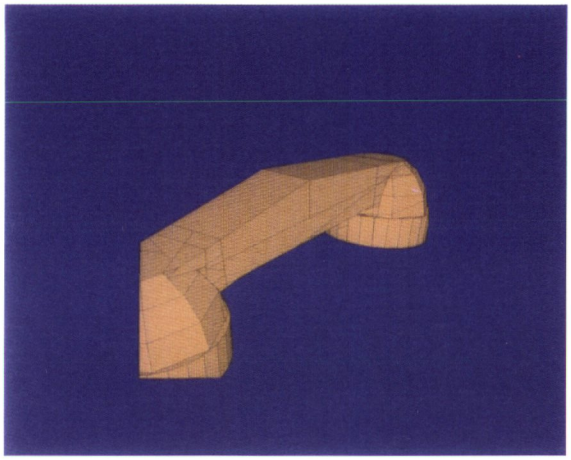
[Willard 82]
Dan E. Willard, "Polygon Retrieval", **SIAM Journal of Computing**, Vol. 11(1), pp. 149-165 (Feb. 1982).



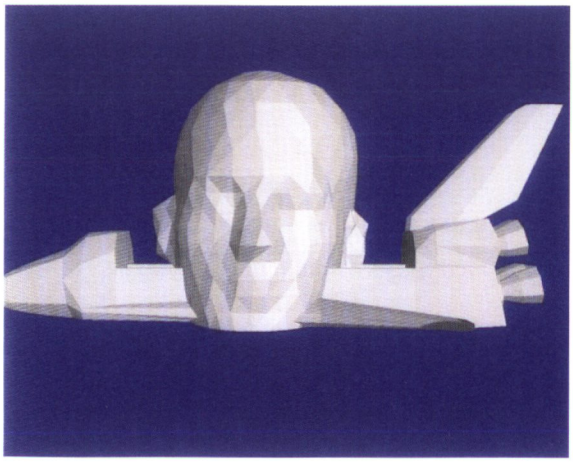
1



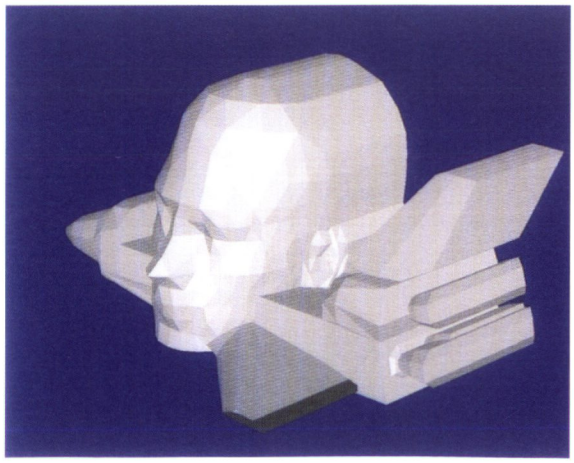
2



3



4



5