Hierarchical Polygon Tiling with Coverage Masks

Ned Greene*

Apple Computer

Abstract

We present a novel polygon tiling algorithm in which recursive subdivision of image space is driven by coverage masks that classify a convex polygon as inside, outside, or intersecting cells in an image hierarchy. This approach permits Warnock-style subdivision with its logarithmic search properties to be driven very efficiently by bit-mask operations. The resulting hierarchical polygon tiling algorithm performs subdivision and visibility computations very rapidly while only visiting cells in the image hierarchy that are crossed by visible edges in the output image. Visible samples are never overwritten. At 512×512 resolution, the algorithm tiles as rapidly as traditional incremental scan conversion, and at high resolution (e.g. 4096×4096) it is much faster, making it well suited to antialiasing by oversampling and filtering. For densely occluded scenes, we combine hierarchical tiling with the hierarchical visibility algorithm to enable hierarchical object-space culling. When we tested this combination on a densely occluded model, it computed visibility on a 4096×4096 grid as rapidly as hierarchical z-buffering [Greene-Kass-Miller93] tiled a 512×512 grid, and it effectively antialiased scenes containing hundreds of thousands of visible polygons. The algorithm requires strict front-to-back traversal of polygons, so we represent a scene as a BSP tree or as an octree of BSP trees. When maintaining depth order of polygons is not convenient, we combine hierarchical tiling with hierarchical z-buffering, resorting to z-buffering only in regions of the screen where the closest object is not encountered first.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - Hidden line/surface removal; I.3.3 [Computer Graphics]: Picture/Image Generation.

Keywords: tiling, coverage mask, antialiasing, visibility, BSP tree, octree, recursive subdivision.

1 INTRODUCTION

Polygon tiling algorithms have been an important topic in computer image synthesis since the advent of raster graphics some two decades ago. Their purpose is to determine which point samples on an image raster are covered by the visible portion of each of the polygons composing a scene. Currently, polygon tiling software running on inexpensive computers can render point-sampled images of simple scenes at interactive rates. The fastest tiling algorithms have been carefully tuned to exploit image-space coherence by using incremental methods wherever possible. However, they fail to exploit opportunities for precomputation and they waste time tiling hidden geometry. There is a need for more efficient tiling algorithms that effectively exploit coherence and precomputation to enable efficient culling of hidden geometry and efficient tiling of visible geometry.

The dominant polygon tiling algorithm in use today is incremental scan conversion. Typically, raster samples on a polygon's perimeter are traversed with an incremental line-tiling algorithm. Edge samples on each intersected scanline define spans within a polygon, which are then traversed pixel-by-pixel, permitting incremental update of shading parameters and, in the case of z-buffering, depth values. Visibility of samples can be determined by a) maintaining a z-buffer and performing depth comparisons [Catmull74], b) traversing primitives back to front and writing every pixel tiled [Foley-et-al90], or c) traversing primitives front to back and overwriting only vacant pixels [Foley-et-al90]. With incremental scan conversion, the cost per pixel tiled is very low because incremental edge and span traversal effectively exploits image-space coherence.

One problem with traditional incremental scan conversion is that it must tile every sample on every primitive, whether or not it is visible, and so it wastes time tiling hidden geometry. This is not a big problem for simple scenes, but for densely occluded scenes it severely impairs efficiency. Ideally, a tiling algorithm should cull hidden geometry efficiently so that running time is proportional to the visible complexity of the scene and independent of the complexity of hidden geometry.

The Warnock subdivision algorithm [Warnock69] approaches this goal, performing logarithmic search for visible tiles in the quadtree subdivision of a polygon. If scene primitives are processed front to back, only visible tiles and their children in the quadtree are visited. Although Warnock subdivision satisfies our desire to work only on visible regions of primitives, the traditional subdivision procedure is relatively slow and consequently, this approach is slower than incremental scan conversion, except for densely occluded scenes. Neither traditional incremental scan conversion nor Warnock subdivision is well suited to tiling scenes of moderate depth complexity.

A second shortcoming of incremental scan conversion is that it spends most of its time tiling edges and spans, traversing these features pixel by pixel, even though all possible tiling patterns for an edge crossing a block of samples can be precomputed and stored as bit masks called *coverage masks*. Then the samples that a convex polygon covers within a block can be quickly found by compositing the coverage masks of its edges. Previously, this technique has been

*Contact author at greene@apple.com,
Apple Computer, 1 Infinite Loop, Cupertino, CA 95014

used to estimate coverage of polygonal fragments within a pixel to accelerate filtering [Carpenter84, Sabella-Wozny83, Fiume-et-al83, Fiume91].

Here we present a polygon tiling algorithm that combines the best features of traditional algorithms. The key innovation that makes this integration possible is the generalization of coverage masks to permit their application to image hierarchies. The generalized masks, which we call *triage coverage masks*, classify cells in the image hierarchy as *inside*, *outside*, or *intersecting* an edge. This enables them to drive Warnock-style subdivision of image space. The result is a hierarchical tiling algorithm that finds visible geometry by logarithmic search, as with the Warnock algorithm, that exploits precomputation of tiling patterns, as with filtering with coverage masks, and that also uses incremental methods to exploit image-space coherence, as with incremental scan conversion. The algorithm efficiently performs high-resolution tiling (e.g. 4096×4096), so it naturally supports high-quality antialiasing by oversampling and filtering. A-buffer-style antialiasing with coverage masks [Carpenter84] is particularly convenient.

For densely occluded scenes we combine hierarchical tiling with the hierarchical visibility algorithm [Greene-Kass-Miller93, Greene-Kass94, Greene95] to permit hierarchical culling of hidden regions of object space. This combination of algorithms enables very rapid rendering of complex polygonal scenes with high-quality antialiasing. The method has been tested and shown to work effectively on densely occluded scenes. On a test scene containing upwards of 167 million replicated polygons, the algorithm computed visibility on a 4096×4096 grid as rapidly as hierarchical z-buffering [Greene-Kass-Miller93] tiled a 512×512 grid.

In §2, we survey previous work on efficient polygon tiling. In §3, we introduce triage coverage masks, and in §4 we present the rendering algorithm in which they are applied. In §5, we discuss how rendering of densely occluded scenes can be accelerated with object-space culling methods. In §6, we discuss strategies for efficiently processing dynamic scenes. In §7, we compare the hierarchical tiling algorithm to hierarchical z-buffering. In §8, we describe hierarchical tiling of polyhedra. In §9, we describe our implementation and show results for both simple and densely occluded scenes. Finally, we present our conclusions in §10.

2 PREVIOUS WORK

2.1 Warnock Subdivision

Our tiling algorithm is loosely based on the Warnock algorithm [Warnock69], a recursive subdivision procedure that finds the quadtree subdivision of visible edges in a scene by logarithmic search. Scene primitives are inserted into a quadtree data structure beginning at the root cell, which represents the whole screen. At each level of subdivision, the algorithm classifies the quadrants of the current quadtree cell as inside, outside, or intersecting the primitive being processed, and only intersected quadrants are subdivided. Quadrants which are entirely covered by one or more primitives are identified, permitting hidden geometry within them to be culled. The Warnock algorithm is actually a family of algorithms based on a common subdivision procedure, and the control structure varies from implementation to implementation [Rogers85]. A typical implementation processes primitives in no particular order, maintains lists of potentially visible primitives at quadtree cells, and expends considerable work performing depth comparisons in order to cull hidden geometry.

When circumstances permit convenient front-to-back traversal of primitives, as with a presorted static polygonal scene, a simpler and more efficient variation of the Warnock algorithm can be employed. In this case, we insert primitives into the quadtree one at a time in front-to-back order. As subdivision proceeds, we mark cells that primitives completely cover as *occupied* and ignore cells that are already occupied, since any geometry that projects to them is known to be hidden. We complete subdivision of one primitive down to the finest level of the quadtree before processing the next. This version of the Warnock algorithm is simpler because it need not maintain lists of primitives or perform depth comparisons. It is more efficient because, unlike the traditional algorithm, it only subdivides cells crossed by edges that are visible in the output image. Our tiling algorithm is based on this variation of the Warnock algorithm, which we will refer to as the *depth-priority Warnock algorithm*. Although Meagher's volume rendering algorithm uses this procedure to tile faces of octree cubes [Meagher82], to the best of our knowledge this variation of the Warnock algorithm has not been applied previously to rendering geometric models. Incidentally, front-to-back traversal of primitives would accelerate Warnock-style subdivision in the error-bounded rendering algorithm described in [Greene-Kass94].

2.2 Coverage Masks

We turn now to reviewing how filtering algorithms exploit precomputation with coverage masks [Carpenter84, Sabella-Wozny83, Fiume-et-al83, Fiume91]. The underlying idea is that all possible tiling patterns for a single edge crossing a grid of raster samples within a pixel can be precomputed and later retrieved, indexed by the points where the edge intersects the pixel's border [Fiume-et-al83, Sabella-Wozny83]. These tiling patterns can be stored as bit masks, permitting samples inside a convex polygon to be determined by ANDING together the coverage masks for its edges. Moreover, if polygons are processed front to back or back to front, visible-surface determination within a pixel can also be performed with bit-mask operations. For example, Carpenter's A-buffer algorithm [Carpenter84] clips polygons to pixel borders, sorts the polygonal fragments front to back, and determines the visible samples on each fragment on a 4×8 grid by compositing coverage masks. The A-buffer algorithm also uses coverage masks to accelerate filtering. For each visible fragment, a single shading value is computed, weighted by the bit count of its mask, and added to pixel color. This shading method efficiently approximates *area sampling* [Catmull78] and it effectively antialiases edges. Abram, Westover, and Whitted advance similar methods that permit jitter, convolution with arbitrary filter kernels, and evaluation of simple shading functions to be performed by table lookup [Abram-et-al85].

3 TRIAGE COVERAGE MASKS

To accelerate polygon tiling, the hierarchical tiling algorithm generalizes coverage masks to operate on image hierarchies, thereby enabling Warnock-style subdivision of image space to be driven by bit-mask operations. A conventional coverage mask for an edge classifies each grid point within a square region of the screen as inside or outside the edge, as shown in figure 1a. In the context of Warnock subdivision, the analogous operation is classifying subcells of an image hierarchy as inside, outside, or intersecting an edge, as shown

in figure 1b for an edge crossing a square containing a 4×4 grid of subcells. We call such masks *triage coverage masks* because the three states that they distinguish correspond to trivial rejection, trivial acceptance, and "do further work." We represent each triage mask as a pair of bit masks, one indicating inside subcells, the other indicating outside subcells, as shown in figures 1c and 1d. We will refer to the bit mask for inside subcells as the "C" mask (for *covered*) and the bit mask for outside subcells as the "V" mask (for *vacant*). We call the intersected subcells the *active* region of the mask, because the corresponding regions of the screen require further work and will later be subdivided. The bit mask for the active region is $A = \tilde{}(C \mid V)$, as shown in figure 1e.[1] In practice, we use 8×8 masks rather than the illustrated 4×4 masks.

The basic tiling and visibility operations performed by conventional coverage masks are (1) finding the mask of a convex polygon from the masks of its edges, and (2) finding the visible samples on a polygon within a pixel by compositing the polygon's mask with the pixel's mask, which represents previously tiled samples. In the context of the hierarchical tiling algorithm, tiling and visibility operations performed by triage masks are entirely analogous, except that compositing is performed recursively on an image hierarchy rather than a single square region of the screen. The image hierarchy is a "coverage pyramid" constructed from both conventional and triage coverage masks, as schematically illustrated in figure 2 (see caption). Operations (1) and (2) for triage masks are easily understood by analogy with conventional coverage masks, as outlined below. See [Greene95] for derivations of the formulas for triage masks and examples illustrating compositing of triage masks.

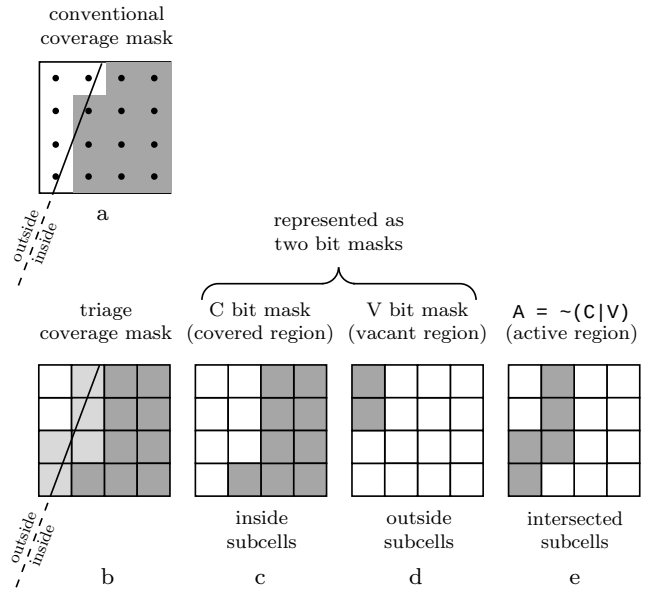

Figure 1: A conventional coverage mask classifies grid points as inside or outside an edge (panel a). A *triage* coverage mask classifies subcells as inside, outside, or intersecting an edge (panel b). We refer to these regions as *covered* (panel c), *vacant* (panel d), and *active* (panel e), respectively. We represent triage masks as the pair of bit masks (C,V) indicating the covered and vacant regions. In practice, we use 8×8 masks rather than 4×4 masks.

```
Tiling a convex polygon into a square region of the
screen using coverage masks.  The existing coverage mask
for a screen cell represents previously tiled polygons,
which are in front of the polygon being tiled.

Conventional Coverage Masks
Existing pixel mask: C
(1)  Find intercepts of edges with pixel border and look
       up edge masks (call them  E1, E2, ... , EN).
     Find mask P of convex polygon from edge masks:
     P = E1 & E2 & ... & EN
(2)  Find mask W of visible samples on polygon within C:
     W = P & ~C
     Update C:
     C' = C | P.

Triage Coverage Masks
Existing triage mask for cell in the coverage pyramid:
  (Cc,Cv) - covered and vacant bit masks
(1)  Find intercepts of edges with cell border and look
       up edge masks ((E1c,E1v), ... ,(ENc,ENv)).
     Find mask (Pc,Pv) of polygon from edge masks:
     Pc = E1c & E2c & ... & ENc
     Pv = E1v | E2v | ... | ENv
(2)  Find mask W of entirely visible cells on polygon
       within (Cc,Cv):
     W = Cv & Pc
     Find mask A of active cells on polygon in (Cc,Cv):
     A = ~(W | Pv | Cc)
     Update (Cc,Cv):
     Cc' = Cc | W
     Cv' = Cv & ~W
     (Note: (Cc,Cv) may also be modified by propagation
       from finer levels.)
```

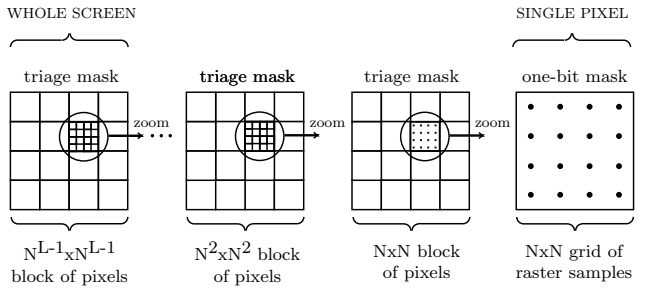

Figure 2: Schematic diagram of a pyramid of $N \times N$ masks with L levels for an image with $N \times N$ oversampling at each pixel. This *coverage pyramid* is built from triage masks, except at the finest level where a conventional one-bit coverage mask is associated with each pixel. In this hierarchical representation of the screen, the C and V bits for each subcell in triage masks indicate whether a square region of the screen is *covered*, *vacant*, or *active*. At the coarsest level, a single triage mask represents the whole screen (left), and at the finest level, a single one-bit mask represents the raster samples within a pixel (right). A four-level pyramid of 8×8 masks corresponds to a 512×512 image with 8×8 oversampling at each pixel. The corresponding diagram for a point-sampled image is the same, except that the masks represent an $N \times N$ block of pixels, an $N^2 \times N^2$ block of pixels, and so forth.

[1] We use standard notation for bit-mask operations: & for bitwise AND, | for bitwise OR, and ~ for bitwise complement.

3.1 Tiling by Recursive Subdivision

Now that the primitive tiling and visibility operations have been described, we are ready to outline the recursive procedure for tiling a convex polygon into the coverage pyramid. Initially, the masks in the coverage pyramid are a hierarchical representation of regions of the image raster that are already occupied by previously tiled polygons. To make the discussion more concrete, the following outline assumes 8×8 oversampling and filtering.

To tile polygon P, we begin by finding the triage mask for each of its edges that crosses the screen by finding its intercepts on the screen border and looking up the corresponding mask in a precomputed table. Then we composite the edge masks according to operation (1) above in order to construct P's triage mask. Next, beginning at the root cell of the coverage pyramid, we composite P's mask with cells in the pyramid, using triage mask operations to distinguish three classes of cells: where P is entirely hidden, where P is entirely visible, and where P's visibility is uncertain, i.e. "active" cells (operation (2)). We ignore cells where P is entirely hidden, we display (or tag) cells where P is entirely visible (mask W), and we recursively subdivide active cells (mask A). During subdivision, edge intercepts used to look up edge masks are computed incrementally. In regions of the screen where P's edges cross vacant or active cells, subdivision continues, ultimately down to all vacant and active pixels crossed by P's edges. At the pixel level, coverage masks in the pyramid are conventional one-bit masks. If we are box filtering, operations may follow the traditional A-buffer algorithm: we find P's visible samples, compute their contribution to pixel value and add it to the accumulation buffer, and then update the pixel's coverage mask. If the status of a pixel changes from vacant or active to active or covered, the status of masks in coarser levels of the pyramid may also change, so whenever this occurs, we propagate coverage information to coarser levels by performing simple bit-mask operations during recursive traversal of the pyramid. When this recursive tiling procedure finishes, all visible samples on P have been tiled and the coverage pyramid has been updated. This procedure is outlined in LISTING 1.

4 RENDERING A SCENE

Now that the procedure for tiling a polygon has been described, we are ready to place it in the context of rendering a frame. But first we describe the underlying data structures: the coverage pyramid, the image array, and the model tree.

4.1 Data Structures

To permit Warnock subdivision to be driven by bit-mask operations, we maintain visibility information about previously tiled polygons in an image-space pyramid of coverage masks. As schematically illustrated in figure 2, a single triage mask represents the whole screen, triage masks at the next level of the pyramid correspond to subcells in the root mask, and so forth. Thus, this *coverage pyramid* is a hierarchical representation of the screen with the C and V bits for each subcell in the triage masks indicating whether a square region of the screen is *covered*, *vacant*, or *active*. Within a covered region, all corresponding samples in the underlying image raster are covered, within a vacant region, all corresponding raster samples are vacant, and within an active region, at least one but not all corresponding raster samples are covered. At the finest level of the pyramid only, we use

LISTING 1 (pseudocode)

```
/*
Recursive subdivision procedure for tiling a convex
polygon P.

After clipping P to the near clipping plane in object
space, if necessary, and projecting P's vertices into
the image plane, we call tile_poly with the root mask of
the mask pyramid, P's edge list, and "level" set to 1.

arguments:
(Cc,Cv): pyramid mask (input and output)
edge_list: P's edges that intersect pyramid mask
level: pyramid level: 1 is root, 2 is next coarsest, etc.
*/

tile_poly((Cc,Cv), edge_list, level)
{
   set active_edge_list to nil

   /* build P's mask (Pc,Pv) */
   Pc = all_ones
   Pv = all_zeros
   for each edge on edge_list {
      find intercepts on square perimeter of mask
      if square is outside edge
      then return /* polygon doesn't intersect mask */
      if edge intersects square, then {
         append edge to active_edge_list
         /* Note: at the pixel level, Ec is a conventional
            coverage mask and Ev = ~Ec */
         look up edge mask (Ec,Ev)
         Pc = Pc & Ec
         Pv = Pv | Ev
      }
   }

   /* make "write" bit mask and update pyramid mask */
   W = Cv & Pc
   Cc = Cc | W
   Cv = Cv & ~W

   if level is the pixel level, then {
      /* filter pixel using coverage mask W */
      /* to perform A-buffer box filtering:
         add bitcount*color to accumulation buffer */
      evaluate shading and update accumulation buffer
      return
   }

   for each TRUE bit in W {
      for each pixel in this square region of screen {
         /* to perform A-buffer box filtering:
            add 64*(polygon color) to accumulation buffer */
         evaluate shading and update accumulation buffer
      }
   }


   /* Recursive Subdivision */

   /* make "active" bit mask */
   A = ~(W | Pv | Cc)
   /* subdivide active subcells */
   for each TRUE bit in A {
      /* call corresponding subcell S
         call its pyramid mask (Sc,Sv) */
      copy all edges on active_edge_list that intersect
        S to S_edge_list
      tile_poly((Sc,Sv), S_edge_list, level+1)
      /* propagate coverage status to coarser levels of
         mask pyramid */
      if Sc is all_ones
      then Cc = Cc | active_bit  /* set covered status  */
      if Sv is not all_ones
      then Cv = Cv & ~active_bit /* clear vacant status */
   }
}
```

conventional one-bit coverage masks to indicate whether or not point samples in the image raster have been covered. If we are oversampling and filtering, each of these one-bit masks corresponds to the 8×8 grid of raster samples within a pixel. The appropriate pyramid for a 512×512 image with 8×8 oversampling at each pixel has four levels, three arrays of triage masks with dimensions 1×1, 8×8, and 64×64, and one 512×512 array of one-bit masks. Alternatively, if we are point sampling rather than filtering, each one-bit mask corresponds to an 8×8 block of pixels. In this case, the pyramid for a 512×512 image would have two arrays of triage masks with dimensions 1×1 and 8×8, and one 64×64 array of one-bit masks.

Memory requirements for the coverage pyramid are very modest. Since the finest level requires only one bit per raster sample and the vast majority of cells in the pyramid are in the finest level, total memory requirements are only slightly more than one bit per raster sample. The actual number of bits per raster sample required for an n-level pyramid lies in the range [1 1/32 1 2/63) for n>1. Note that a z-buffer requires a great deal more memory because it stores a depth value for each raster sample.

The other image-space data structure that our algorithm requires is an image array with an element for each color component at each pixel. If we perform A-buffer-style filtering [Carpenter84], shading contributions from 64 subpixel samples accumulate in each array element. Thus, elements in this *accumulation buffer* require considerable depth. We use 16 bits per pixel per color channel. When filtering with a convolution kernel that overlaps multiple pixels, we store color components as floating-point values in the accumulation buffer. If no filtering is performed, pixel values do not accumulate, so a conventional image array is employed.

Now for representing the model. Our algorithm requires front-to-back traversal of polygons in the scene, so we represent the scene as a binary space partitioning tree (BSP tree) [Fuchs-Kedem-Naylor80], which permits very efficient traversal in depth order. Strategies for handling dynamic scenes are discussed in §6.

4.2 Precomputation Step

In a precomputation step, we build a BSP tree for the model. We also build lookup tables for both conventional and triage coverage masks. In building mask tables, we divide the perimeter of a canonical square into some number of equal intervals (e.g. 64) and create an entry in a two-dimensional table for each pair of intervals not lying on a common edge. Once this table has been constructed, to obtain the mask for an arbitrary edge we determine which two intervals it crosses and look up the corresponding table entry. To conserve storage, we can use the same table entry for edges with opposite directions, because complementing the (C,V) bit masks in a triage mask corresponds to reversing an edge. Hierarchical tiling depends on accurate classification of *vacant* and *covered* regions in triage masks, so we construct them with the following conservative procedure. The endpoints of the pair of intervals used to index a coverage mask define a quadrilateral. Any subcells intersected by the quadrilateral are classified *active*, guaranteeing that cells classified *covered* are completely covered and cells classified *vacant* are completely vacant.

4.3 Generating a Frame

We begin a frame by clearing the accumulation buffer and the coverage pyramid. We traverse polygons in the model's BSP tree in front-to-back order. We clip each polygon to the front clipping plane, if necessary, and project its vertices into the image plane. There is no need to preserve depth information. Before tiling a polygon, we first determine whether its bounding box is visible. If this procedure fails to prove that the polygon is hidden, we then tile it into the smallest enclosing cell in the coverage pyramid using the procedure outlined in LISTING 1. In regions of the screen where the polygon is visible, this procedure updates pixel values in the image buffer and updates coverage status in the coverage pyramid. After all polygons have been processed, the scene is complete and we display the image buffer.

4.4 Other Filtering Methods

We have already discussed A-buffer-style filtering by *area sampling*, a term used to describe convolution of visible samples with a pixel-sized box filter [Catmull78]. Abram, Westover, and Whitted extended coverage-mask techniques to include jitter, table-driven convolution with arbitrary filter kernels, and evaluation of simple shading functions by table lookup [Abram-et-al85]. All of these methods are compatible with hierarchical tiling. To perform table-driven convolution, the contribution of each subpixel sample to neighboring pixels is precomputed and stored in a table of filtering coefficients. For some simple shading functions, the contribution of arbitrary collections of samples can be stored as precomputed coefficients which enables, for example, efficient byte-by-byte processing of coverage masks. We use this method when filtering 3×3 pixel neighborhoods with a one-pixel radius cosine-hump kernel.

4.5 Point Sampling

Modifying the algorithm to produce point-sampled rather than filtered images is straightforward. In this case, each mask at the finest level of the pyramid corresponds to an 8×8 block of pixels. So for each TRUE subcell in the "W" mask (see pseudocode), we evaluate the shading function at the corresponding pixel and write the result to the image buffer. Since pixel values correspond to point samples, color values do not accumulate, so we use a conventional image array rather than an accumulation buffer. Note that it is not necessary to clear the image array at the beginning of a frame. Instead, after tiling all scene polygons, we composite a screen-sized polygon of the desired background color (or texture) with the root mask, thereby tiling all remaining vacant pixels in the image.

5 HIERARCHICAL OBJECT-SPACE CULLING

Because of its ability to cull hierarchically in image space, the hierarchical tiling algorithm processes densely occluded scenes much more efficiently than conventional tiling methods, which must traverse all hidden geometry pixel by pixel. Nonetheless, it must still consider every polygon in a scene, doing some work even on those that are entirely hidden. To avoid this behavior, we integrate our algorithm with the hierarchical visibility algorithm [Greene-Kass-Miller93, Greene-Kass94, Greene95] to enable hierarchical object-space culling of hidden regions of the model. This can be done by substituting hierarchical tiling for z-buffering in the

hierarchical z-buffer algorithm of [Greene-Kass-Miller93], although this requires some changes in both the object-space and image-space hierarchies. In image space, instead of using a z-pyramid of depth samples to maintain visibility information, we use a coverage pyramid. In object space, we modify the octree to permit strict front-to-back traversal of polygons. Note that the z-buffer algorithm traverses octree cubes in front-to-back order, but not the primitives contained within them. And since octree cubes are nested, it is not sufficient to simply organize the primitives inside each cube into a BSP tree. Instead we use the following algorithm for building an *octree of BSP trees* that permits strict front-to-back traversal.

5.1 Building an Octree of BSP Trees

Starting with a root cube which bounds model space, we insert polygons one at a time into the cube. If the polygon count in the cube reaches a specified threshold (e.g. 30), we subdivide the cube into eight octants and insert each of its polygons into each octant that it intersects, clipping to the cube's three median planes. When all polygons in the scene have been inserted into the root cube and propagated through the tree, we have an octree where all polygons are associated only with leaf nodes, thereby circumventing the ordering problem caused by nesting. The last step is to organize the polygons in each leaf node of the octree into a BSP tree [Foley-et-al90]. Now scene polygons can be traversed in strict front-to-back order by traversing octree cubes front to back and traversing their BSP trees front to back.

5.2 Combining Hierarchical Tiling with Hierarchical Visibility

Now that we have established how to traverse scene polygons in front-to-back order, combining hierarchical tiling with the basic hierarchical visibility algorithm is straightforward. As with hierarchical z-buffering, we traverse octree cubes in front-to-back order, testing them for visibility and culling those that are hidden. As with hierarchical z-buffering, we determine whether a cube is visible by tiling it, stopping if a visible sample is found. Note that it is only necessary to tile a cube's polygonal silhouette (unless it intersects the front clipping plane), rather than tiling its front faces. By comparison, z-buffering often needs to tile three faces of a cube to establish its visibility. To test cube silhouettes for visibility, we modify the tiling procedure of LISTING 1 to report visibility status, returning TRUE whenever a polygon's mask indicates that it covers a vacant subcell or a vacant grid point in the image raster. Once we have established that an octree cube is visible, we traverse the polygons in its BSP tree in front-to-back order, tiling them into the coverage pyramid. When we finish traversing the octree, all visible polygons have been tiled and the image is complete.

This version of the hierarchical visibility algorithm has very efficient traversal properties in both object-space and image-space. Like the hierarchical z-buffer algorithm, in object space the algorithm only visits visible octree nodes and their children, and it only renders polygons that are in visible octree nodes. In image space, when tiling polygons into the coverage pyramid, hierarchical tiling only visits cells that are crossed by visible edges in the output image. Visible samples are never overwritten. As a result of these properties, this variation of the hierarchical visibility algorithm is very efficient at both culling hidden geometry and tiling visible geometry.

If a hardware graphics accelerator is available to perform shading operations such as texture mapping, we can perform visibility operations with software and shading with hardware. We would use the usual hierarchical tiling algorithm to maintain the coverage pyramid and perform object-space culling, and we would render visible polygons with the graphics accelerator, using an accumulation buffer, if available, to perform antialiasing [Haeberli-Akeley90]. This would be a fast way to produce texture-mapped images of densely occluded scenes.

6 HANDLING DYNAMIC SCENES

One weakness of the hierarchical tiling algorithm is that it requires strict front-to-back traversal of polygons. This does not present a problem for a static model, since it may be represented as a BSP tree [Fuchs-Kedem-Naylor80], and if only a relatively small number of polygons are moving, the tree can be efficiently maintained [Naylor92a]. However, in scenes with numerous moving polygons, maintaining depth order can impose a severe computational burden. Here we consider two different methods that address this problem.

6.1 Lazy Z-Buffering

The following "lazy z-buffering" algorithm is an attractive alternative whenever at least part of the model can be conveniently traversed in approximate front-to-back order. For convenience, the following discussion assumes that we are oversampling and box-filtering. With this variation of hierarchical tiling, we make the following changes to the basic algorithm. For every cell in the coverage pyramid, we maintain *znear* and *zfar* depth values for all potentially visible polygons thus far encountered that intersect the cell. Instead of automatically culling a portion of a polygon that intersects a covered cell, it is culled only if it lies behind the cell's zfar value. At a pixel, we assume that fragments arrive in an order that permits tiling with coverage masks, i.e., one or more non-overlapping fragments cover all of the pixel's samples before any other fragments arrive. These conditions are easily monitored using the pixel's coverage mask and znear/zfar values. Unless and until a fragment violating the conditions arrives, we perform filtering like the usual algorithm, adding shading contributions to the accumulation buffer and updating the pixel's coverage mask. We also cache information about each fragment in case we need it later. If and when the conditions are violated, we discard the current accumulated color value for the pixel and revert to ordinary z-buffering, allocating the memory required for storing color and depth at each subpixel sample, and then tiling the cached fragments. This produces the same image samples as if we had been maintaining an oversampled z-buffer all along. The last step after all polygons in the scene have been tiled is to filter the z-buffered pixels. This procedure produces the same image as hierarchical tiling would have produced if polygons had been traversed in depth order.

This simple strategy exploits whatever depth coherence is in the scene being processed. If polygons are mostly in front-to-back order, lazy z-buffering will not do much more work than the usual hierarchical tiling algorithm. This would occur, for example, if a few small dynamic objects were positioned in front of a static background model that was traversed in depth order. In the worst case, when frontmost objects are never processed first, lazy z-buffering does only slightly more work than hierarchical z-buffering.

Thing Being Compared	Hierarchical Tiling	Hierarchical Z-Buffering
object-space hierarchy	BSP tree / octree of BSP trees	octree
image-space hierarchy	pyramid of coverage masks	z-pyramid
front-to-back polygon traversal required?	yes	no
visibility information per raster sample	< 1 2/63 coverage-mask bits	Z (usu. 24-32 bits)
color information per raster sample	none	RGB (usu. 24-36 bits)
type of output-image buffer	accumulation (deep)	standard
need to store coverage-mask LUTs?	yes	no
pixel overwrite?	no	yes
mask support for filtering built-in?	yes	no
identifies *covered* image-pyramid cells?	yes	no

Table 1: Some points of comparison between hierarchical polygon tiling and hierarchical z-buffering.

6.2 Merging Octrees

For polygonal scenes consisting of independently moving rigid bodies, another strategy can be employed that guarantees front-to-back traversal of polygons, permitting us to render polygons with the standard hierarchical tiling procedure. According to this method, each rigid body is represented as an octree of BSP trees. To render a frame, we simultaneously traverse all octrees front to back, culling any octree cubes which are hidden by the coverage pyramid, and using the following strategy to synchronize traversal of octrees. For each octree, we determine the current frontmost leaf cube and then determine the frontmost leaf cube of all octrees. If this single frontmost cube does not intersect a leaf cube in any other octree, we can safely render its BSP tree. If this cube does intersect other leaf cubes, we clip their polygons to the frontmost cube, insert the clipped fragments into the frontmost cube's BSP tree, and then render that BSP tree. This procedure ultimately will cull or render all octree leaf nodes, whereupon rendering of the scene is finished. This procedure for rendering dynamic scenes is nearly as fast as the standard hierarchical tiling algorithm, except for the time spent merging octree leaf nodes. Although merging operations can require considerable computation, for many scenes merging will only rarely be required, and in such cases this algorithm will run efficiently.

7 HIERARCHICAL TILING VERSUS HIERARCHICAL Z-BUFFERING

Table 1 summarizes some points of comparison between hierarchical polygon tiling and hierarchical z-buffering. As the table points out, hierarchical tiling requires strict front-to-back traversal of polygons, which complicates the object-space hierarchy, assuming that we are maintaining an octree of BSP trees to enable object-space culling. Another point in favor of hierarchical z-buffering is that it does not need to build or store lookup tables for coverage masks. The other points of comparison strongly favor hierarchical tiling. One big advantage is that its memory requirements are much less. Whereas hierarchical z-buffering needs to store depth and color information for each raster sample, hierarchical tiling only needs to store slightly more than one bit of coverage information for each raster sample. The resulting memory savings can be very substantial. In fact, if we are rendering a 512×512 image with 8×8 oversampling at each pixel, hierarchical tiling requires only about 3.7% of the image memory required for z-buffering. Other points in favor of hierarchical tiling are that it never overwrites visible samples, it has built-in support for filtering with coverage masks, and it facilitates exploiting image-space coherence by identifying regions of the image-space pyramid that are completely covered by individual polygons.

8 TILING POLYHEDRA

Hierarchical tiling with coverage masks can also be applied to Warnock subdivision in three dimensions to tile convex polyhedra into a voxel grid. In this case, 64-bit triage masks would classify cells within a 4×4×4 subdivision of a cube as inside, outside, or intersecting a plane. The triage mask for a convex polyhedron within a cube would be obtained by compositing the triage masks of its face planes. The recursive subdivision procedure for tiling a polyhedron into a 3D pyramid of coverage masks would be analogous to hierarchical polygon tiling, and it would only visit cells in the pyramid that are intersected by the polyhedron's faces. The speed and modest memory requirements of this volume tiling algorithm make it an attractive alternative to traditional methods [Kaufman86].

9 IMPLEMENTATION AND RESULTS

Our implementation of hierarchical polygon tiling is programmed in C and renders either point-sampled or filtered images of scenes composed of flat-shaded convex polygons. Our polygon tiling program follows the pseudocode outline, except that we tile a polygon into the smallest enclosing cell in the coverage pyramid after first testing its bounding box for visibility, as described in §4.3. As described in §3 and §4, filtering is performed by box filtering according to the A-buffer method, or by table-driven convolution with a one-pixel radius cosine-hump kernel. In the latter case, kernel coefficients are precomputed for all byte patterns and accessed by table lookup for each non-zero byte within a polygon's coverage mask at a pixel. Color components in the accumulation buffer are represented as 16-bit integer values when box filtering, and as 32-bit floating-point values when filtering with a cosine-hump kernel. Tables of coverage masks are constructed with 64 intervals along each edge of the bounding square. One-bit coverage masks for filtering pixels are constructed with jitter, using random placement of raster samples within the corresponding sub-pixel square [Dippé-Wold85, Cook86]. All of the following tests were performed on a SGI Indigo2 with a 75 megahertz R8000 processor, which performs atomic 64-bit mask operations.

To compare the efficiency of hierarchical tiling to traditional incremental scan conversion for tiling simple polygonal scenes, we employed the color-cube model of figure 5,

composed of 192 presorted front-facing squares. We rendered this model with hierarchical tiling and with a back-to-front "painter's" algorithm [Foley-et-al90]. The painter's algorithm maintained a color triplet for each point in the image raster and performed tiling by incremental scan conversion, overwriting the image at every pixel encountered. On a 512×512 grid, hierarchical tiling tiled the color-cube model approximately ten percent faster than the painter's algorithm (.087 seconds versus .097 seconds). At higher resolution, the speed advantage of hierarchical tiling was much more pronounced. For example, hierarchical tiling took .357 seconds to tile the model on a 4096×4096 grid and produce the 512×512 box-filtered image of figure 5. By comparison, the painter's algorithm took 5.3 times longer (1.91 seconds) to tile this scene on a 2048×2048 grid without filtering (our Indigo didn't have enough memory to render a 4096×4096 RGB image). By timing the painter's algorithm at various resolutions, we found that it was only able to tile a 910×910 grid in the .357 seconds it took hierarchical tiling to tile and filter the image of figure 5. This example illustrates that for software tiling at sufficient resolution to enable high-quality antialiasing by oversampling and filtering, hierarchical tiling is much more efficient than traditional incremental scan conversion, even for simple scenes.

To test the effectiveness of hierarchical tiling on densely occluded scenes we integrated hierarchical tiling with hierarchical visibility as described in §5, performing tiling of both model polygons and octree-cube silhouettes with the hierarchical tiling method. For a test model, we used a version of the modular office building described in [Greene-Kass-Miller93]. We built an octree of BSP trees for the repeating module using the method described in §5.1, each BSP tree containing approximately 16,000 quadrilaterals. We replicated this octree within the shell of a 408-story building resembling the Empire State Building to create a model consisting of approximately 167 million replicated quadrilaterals. Figures 4, 6, and 7 show various views of this model.

To compare the relative speed of hierarchical tiling and hierarchical z-buffering, we rendered animation of a building walk-through. We found that hierarchical tiling was able to perform tiling on a 4096×4096 grid and produce box-filtered 512×512 frames as fast as hierarchical z-buffering produced 512×512 point-sampled frames. On viewing the animation produced with the z-buffer algorithm, we observed considerable aliasing as expected. By comparison, we observed high-quality antialiasing with the box-filtered animation generated with hierarchical tiling.

Next, we compared the speed of various rendering options. Hierarchical tiling took 3.21 seconds to tile the scene of figure 4 on a 4096×4096 grid and produce the pictured box-filtered image. When we rendered this same box-filtered image without using the bounding-box culling method of §4.3 and instead tiled all polygons into the root cell of the coverage pyramid, rendering took 1.28 seconds longer, indicating that the bounding-box culling strategy provides significant acceleration. Next we rendered the scene of figure 4 with higher-quality antialiasing using cosine-hump filtering within a 3×3-pixel neighborhood. With this filtering method, it took 4.42 seconds to tile the scene on a 4096×4096 grid and produce the filtered image. Finally, we used the point-sampling variation of hierarchical tiling to render a 512×512 image of the scene, which took 1.71 seconds.

To compare the algorithmic efficiency of hierarchical tiling to hierarchical z-buffering, we constructed "work images" that show the number of times during frame generation that each cell in the coverage pyramid is visited (not counting subpixel samples), with an access to a coarser-than-pixel cell being amortized over the corresponding window of the screen.[2] Work images show the "depth complexity" of the visibility computation and indicate where the algorithm is working hardest [Greene95, Greene-Kass94, Greene-Kass-Miller93]. An average intensity of one in a work image means that, an average, only a single pyramid cell is accessed in the coverage pyramid during visibility operations for each pixel in the output image. With hierarchical tiling, except for very complex scenes or finely tessellated models, average intensity is usually less than one because visibility at most pixels is established at a coarser level in the hierarchy. For example, for the simple model of figure 5, an average of only .123 cells in the coverage pyramid are traversed per pixel in the output image.

Figure 3 shows log-scale work images corresponding to figure 4. Left to right, the images show work tiling cube silhouettes during visibility tests (.09 cells visited per pixel, on average), work tiling model polygons into the coverage pyramid (1.01 cells visited per pixel, on average), and the sum of these two images, showing total work performed on tiling (1.10 cells visited per pixel, on average). In other words, hierarchical tiling visited an average of only 1.10 cells in the coverage pyramid for each pixel in the 512×512 output image, even though tiling and filtering were performed on a 4096×4096 grid. Far fewer cells in the image pyramid are visited with hierarchical tiling than with hierarchical z-buffering [Greene95] because it only visits cells in the image pyramid that are crossed by visible edges in the output image. When we performed a motion test on the scene of figure 4, we found that the number of pyramid cells visited was approximately one for frames rendered with hierarchical tiling and approximately three for frames rendered with hierarchical z-buffering. The lower figure for hierarchical tiling is particularly impressive considering that it resolves visibility at 64 times as many raster samples and many more polygons are visible. Of course the depth complexity of visibility computations for the scene of figure 4 is far lower for both hierarchical tiling and hierarchical z-buffering than for naive z-buffering, which visits each pixel dozens of times on average [Greene95].

To explore the limits of hierarchical tiling to effectively filter images of very complex scenes, we rendered a motion sequence in which the camera flies around and through the 408-story model of the Empire State Building [Greene96]. Figures 6 and 7 are 512×512 frames from this animation, which was produced by tiling on a jittered 4096×4096 grid and filtering with a cosine-hump kernel as previously described. From the viewpoint of figure 6, this scene poses a formidable challenge to effective filtering, since approximately 765,000 polygons are visible, and dozens of polygons are visible within some pixels. Nonetheless, we observed high-quality antialiasing in the motion sequence. Jittering of sub-pixel samples effectively converted aliasing to noise [Dippé-Wold85, Cook86], which was noticeable only in frames having hundreds of thousands of visible polygons. For this motion sequence, we observed subtle patterned aliasing artifacts when the same jitter pattern was employed at all pixels, a problem that was overcome by using several different jitter patterns. To reduce temporal aliasing we rendered each video field separately and to reduce flicker we

[2] Our accounting of work done on visibility does not include clearing of the coverage pyramid. Clearing the pyramid at the beginning of a frame visits each pyramid cell once, but this is not necessary if a "lazy clearing" strategy is employed.

applied a 1-4-6-4-1 filter to every other scanline (not applied to figure 6 or 7). With cosine-hump filtering and multiple jitter patterns, rendering times were 5.15 minutes for figure 6 and 34 seconds for figure 7, which has approximately 81,300 visible polygons. We also recorded the motion sequence with box filtering, but observed noticeably worse image quality, particularly the characteristic "ropyness" of area sampling. When box filtering with a single jitter pattern, rendering time for the scene of figure 6 was 4.28 minutes. We also rendered the version of this model shown in figure 1 of [Greene-Kass94], which took the error-bounded rendering algorithm described in that article one hour to produce on a 50-megahertz workstation. By comparison, this same scene took hierarchical tiling 34 seconds to render. These examples illustrate that hierarchical tiling with object-space culling can produce high-quality animation of very complex scenes in reasonable frame times. Adding a level to the coverage pyramid would permit the algorithm to accurately filter even more complex scenes.

10 CONCLUSION

Warnock subdivision with its elegant simplicity and logarithmic search properties endures as one of the great computer-graphics algorithms. Although polygon tiling by Warnock subdivision is well known, it has rarely been used in practice due to the inefficiency of the traditional subdivision procedure. Here we have shown that Warnock-style subdivision can be driven very efficiently with *triage coverage masks*. The resulting hierarchical polygon tiling algorithm is very efficient, visiting only cells in the image hierarchy that are crossed by visible edges in the output image and never overwriting a visible image sample. At high resolution, hierarchical tiling is much faster than traditional incremental scan conversion, so it is well suited to antialiasing by oversampling and filtering. Moreover, hierarchical tiling with object-space culling can process densely occluded scenes extremely efficiently, considerably faster than hierarchical z-buffering, while facilitating high-quality filtering. Although the practicality of the basic algorithm for dynamic scenes is constrained by the requirement that polygons be traversed front to back, whenever at least part of the model can be traversed in approximate front-to-back order, "lazy z-buffering" helps to overcome this shortcoming. The algorithm is compact, straightforward to implement, and has very modest memory requirements. In short, hierarchical tiling offers the prospect of generating high-quality animation at reasonable frame rates with modest computing resources.

11 Acknowledgments

Gavin Miller suggested the method of combining hardware and software tiling discussed in §5.2. Gavin also contributed to making the test model of figure 4, as did Eric Chen and Steve Rubin. A discussion with Piter van Zee was my impetus for working out the algorithm for merging octrees presented in §6.2. Finally, I gratefully acknowledge Siggraph reviewer #4 for a thoughtful critique of this article.

References

[Abram-et-al85] G. Abram, L. Westover, and T. Whitted, "Efficient Alias-Free Rendering using Bit-Masks and Look-Up Tables," *Proceedings of SIGGRAPH '85*, July 1985, 53–59.

[Carpenter84] L. Carpenter, "The A-buffer, an Antialiased Hidden Surface Method," *Proceedings of SIGGRAPH '84*, July 1984, 103–108.

[Catmull74] E. Catmull, "A Subdivision Algorithm for Computer Display of Curved Surfaces," PhD Thesis, Report UTEC-CSc-74-133, Computer Science Dept., University of Utah, Salt Lake City, Utah, Dec. 1974.

[Catmull78] E. Catmull, "A Hidden-Surface Algorithm with Anti-Aliasing," *Proceedings of SIGGRAPH '78*, Aug. 1978, 6–11.

[Cook86] R. Cook, "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics*, Jan. 1986, 51–72.

[Dippé-Wold85] M. A. Z. Dippé and E. H. Wold, "Antialiasing through Stochastic Sampling," *Proceedings of SIGGRAPH '85*, July 1985, 69–78.

[Fiume-et-al83] E. Fiume, A. Fournier, and L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer," *Proceedings of SIGGRAPH '83*, July 1983, 141–150.

[Fiume91] E. Fiume, "Coverage Masks and Convolution Tables for Fast Area Sampling," *Graphical Models and Image Processing*, 53(1), Jan. 1991, 25–30.

[Foley-et-al90] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics, Principles and Practice*, 2nd edition, Addison-Wesley, Reading, MA, 1990.

[Fuchs-Kedem-Naylor80] H. Fuchs, J. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," *Proceedings of SIGGRAPH '80*, June 1980, 124–133.

[Greene-Kass-Miller93] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," *Proceedings of SIGGRAPH '93*, July 1993, 231–238.

[Greene-Kass94] N. Greene and M. Kass, "Error-Bounded Antialiased Rendering of Complex Environments," *Proceedings of SIGGRAPH '94*, July 1994, 59–66.

[Greene95] N. Greene, "Hierarchical Rendering of Complex Environments," PhD Thesis, Univ. of California at Santa Cruz, Report No. UCSC-CRL-95-27, June 1995.

[Greene96] N. Greene, "Naked Empire," ACM Siggraph Video Review Issue 115: The Siggraph '96 Electronic Theater, August 1996.

[Haeberli-Akeley90] P. Haeberli and K. Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering," *Proceedings of SIGGRAPH '90*, Aug. 1990, 309–318.

[Kaufman86] A. Kaufman, "3D Scan Conversion Algorithms for Voxel-Based Graphics," *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, Oct. 1986, 45–75.

[Meagher82] D. Meagher, "The Octree Encoding Method for Efficient Solid Modeling," PhD Thesis, Electrical Engineering Dept., Rensselaer Polytechnic Institute, Troy, New York, Aug. 1982.

[Naylor92a] B. Naylor, "Interactive Solid Geometry Via Partitioning Trees," *Proceedings of Graphics Interface '92*, May 1992, 11–18.

[Naylor92b] B. Naylor, "Partitioning Tree Image Representation and Generation from 3D Geometric Models," *Proceedings of Graphics Interface '92*, May 1992, 201–212.

[Rogers85] D. Rogers, *Procedural Elements for Computer Graphics,* McGraw-Hill, New York, 1985.

[Sabella-Wozny83] P. Sabella and M. Wozny, "Toward Fast Color-Shaded Images of CAD/CAM Geometry," *IEEE Computer Graphics and Applications*, 3(8), Nov. 1983, 60–71.

[Teller92] S. Teller, "Visibility Computations in Densely Occluded Polyhedral Environments," PhD Thesis, Univ. of California at Berkeley, Report No. UCB/CSD 92/708, Oct. 1992.

[Warnock69] J. Warnock, "A Hidden Surface Algorithm for Computer Generated Halftone Pictures," PhD Thesis, Computer Science Dept., University of Utah, TR 4-15, June 1969.

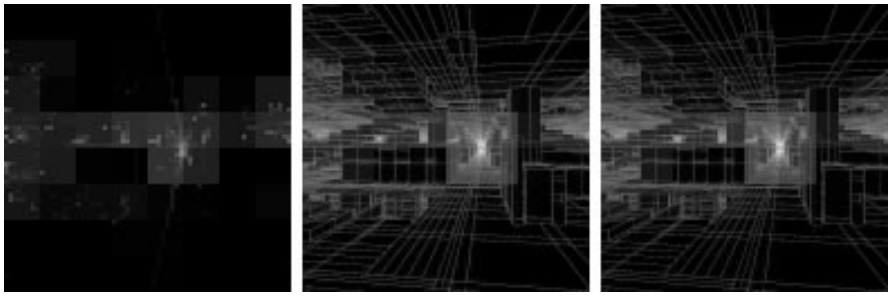

Figure 3: Log-scale work images showing the number of times that cells in the coverage pyramid were visited while tiling the frame of figure 4. These images depict the "depth complexity" of the visibility computation, showing where the algorithm is working hardest.
Left: work tiling cubes: .09 cells visited per pixel (avg)
Middle: work tiling polygons: 1.01 cells visited per pixel (avg)
Right: total work on tiling: 1.10 cells visited per pixel (avg)

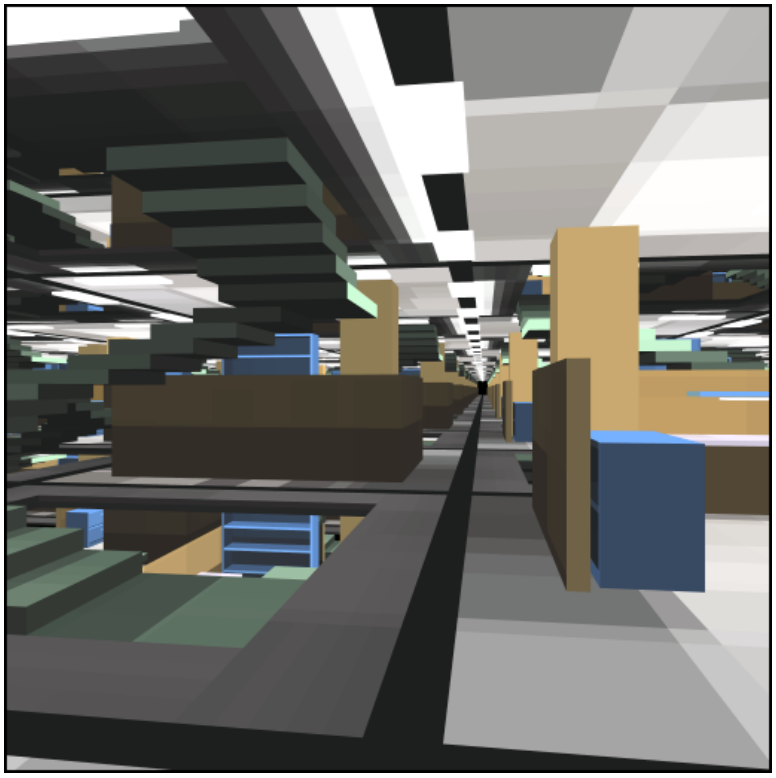

Figure 4: Interior view of the Empire State Building model. Hierarchical tiling took 3.21 seconds to tile this scene on a 4096×4096 grid and produce this 512×512 box-filtered image (75 Mhz processor).

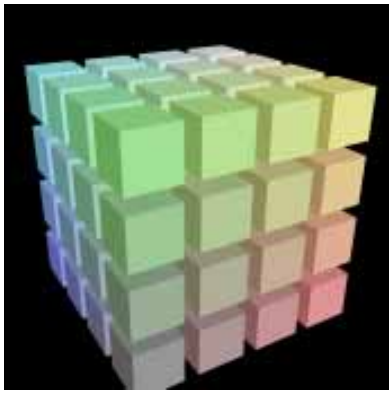

Figure 5: Hierarchical tiling took .36 seconds to tile this simple model on a 4096×4096 grid and produce this 512×512 box-filtered image (75 Mhz processor).

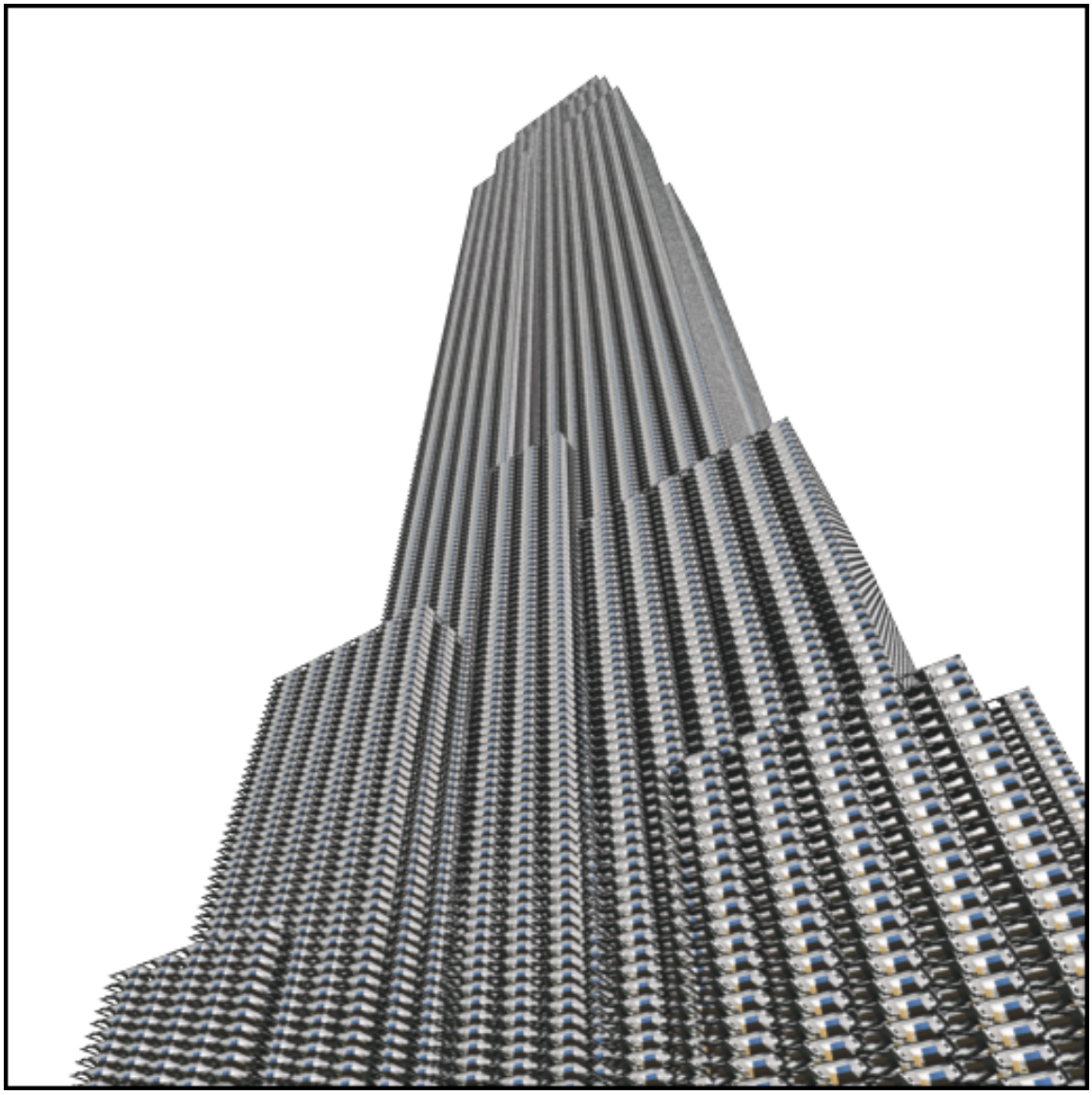

Figure 6: A frame from "Naked Empire," animation produced for the Siggraph '96 Electronic Theater [Greene96]. The model of this 408-story building consists of approximately 167 million quadrilaterals, 765,000 of which are visible in this frame. This 512×512 frame was produced by tiling and filtering on a jittered 4096×4096 grid. Jitter converted aliasing to noise, which is evident in complex regions of the image. Rendering took 5.15 minutes on a 75 Mhz processor.

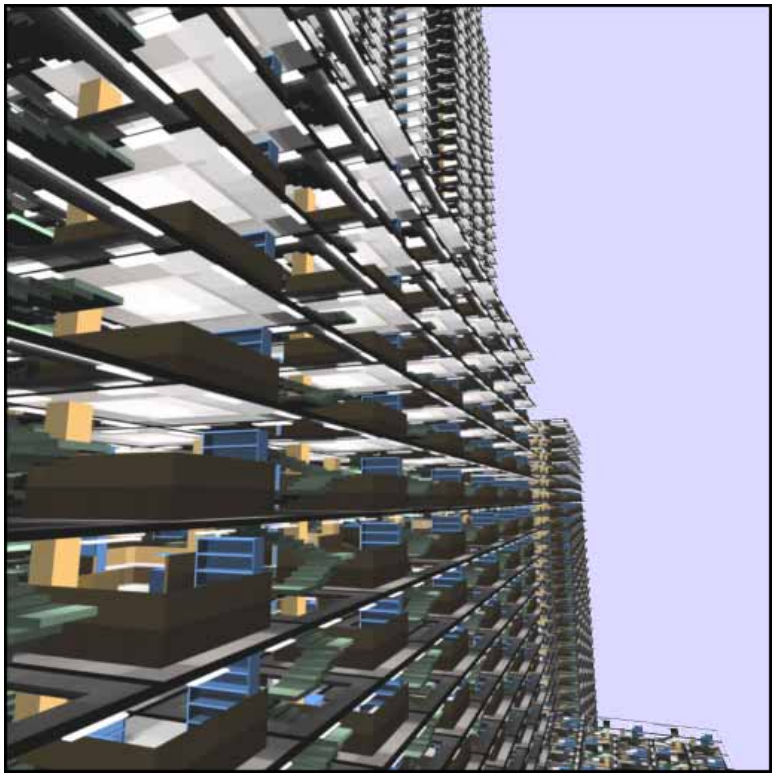

Figure 7: Another frame from "Naked Empire." Note that the building model has no outer shell, making it possible to see deep inside. Rendering time for this frame was 34 seconds (75 Mhz processor).