Visibility Sorting and Compositing without Splitting for Image Layer Decompositions

John Snyder and Jed Lengyel
Microsoft Research

Abstract

We present an efficient algorithm for visibility sorting a set of moving geometric objects into a sequence of image layers which are composited to produce the final image. Instead of splitting the geometry as in previous visibility approaches, we detect mutual occluders and resolve them using an appropriate image compositing expression or merge them into a single layer. Such an algorithm has many applications in computer graphics; we demonstrate two: rendering acceleration using image interpolation and visibility-correct depth of field using image blurring.

We propose a new, incremental method for identifying mutually occluding sets of objects and computing a visibility sort among these sets. Occlusion queries are accelerated by testing on convex bounding hulls; less conservative tests are also discussed. Kd-trees formed by combinations of directions in object or image space provide an initial cull on potential occluders, and incremental collision detection algorithms are adapted to resolve pairwise occlusions, when necessary. Mutual occluders are further analyzed to generate an image compositing expression; in the case of non-binary occlusion cycles, an expression can always be generated without merging the objects into a single layer. Results demonstrate that the algorithm is practical for real-time animation of scenes involving hundreds of objects each comprising hundreds or thousands of polygons.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation - Display algorithms.
Additional Keywords: visibility sorting, compositing, nonsplitting layered decomposition, occlusion cycle, occlusion graph, sprite, kd-tree.

1 Introduction

This paper addresses the problem of how to efficiently sort dynamic geometry into image layers. Applications include:

1. image-based rendering acceleration – by using image warping techniques rather than re-rendering to approximate appearance, rendering resources can be conserved [Shade96,Schaufler96,Torborg96, Lengyel97].
2. image stream compression – segmenting a synthetic image stream into visibility-sorted layers yields greater compression by exploiting the greater coherence present in the segmented layers [Wang94,Ming97].
3. fast special effects generation – effects such as motion blur and depth-of-field can be efficiently computed via image post-processing techniques [Potmesil81,Potmesil83,Max85,Rokita93]. Visibility sorting corrects errors due to the lack of information on occluded surfaces in [Potmesil81,Potmesil83,Rokita93] (see [Cook84] for a discussion of these errors), and uses a correct visibility sort instead of the simple depth sort proposed in [Max85].
4. animation playback with selective display/modification – by storing the image layers associated with each object and the image compositing expression for these layers, layers may be selectively added, removed, or modified for fast preview or interactive playback. Unchanged layers require no re-rendering.
5. incorporation of external image streams – hand-drawn character animation, recorded video, or off-line rendered images can be inserted into a 3D animation using a geometric proxy which is ordered along with the 3D synthetic elements, but drawn using the 2D image stream.
6. rendering with transparency – while standard z-buffers fail to properly render arbitrarily-ordered transparent objects, visibility sorting solves this problem provided the (possibly grouped) objects themselves can be properly rendered.
7. fast hidden-line rendering – by factoring the geometry into sorted layers, we reduce the hidden line problem [Appel67,Markosian97] into a set of much simpler problems. Rasterized hidden line renderings of occluding layers simply overwrite occluded layers beneath.
8. rendering without or with reduced z-buffer use – the software visibility sort allows elimination or reduced use of hardware z-buffers. Z-buffer resolution can also be targeted to the extent of small groups of mutually occluding objects, rather than the whole scene's.

To understand the usefulness of visibility sorting, we briefly focus on rendering acceleration. The goal is to render each coherent object at the appropriate spatial and temporal resolution and interpolate with image warps between renderings. Several approaches have been used to compose the set of object images. We use pure image sprites without z information, requiring software visibility sorting [Lengyel97]. Another approach caches images with sampled (per-pixel) z information [Molnar92,Regan94,Mark97, Schaufler97], but incurs problems with antialiasing and depth uncovering (disocclusion). A third approach is to use texture-mapped geometric impostors like single quadrilaterals [Shade96,Schaufler96] or polygonal meshes [Maciel95,Sillion97]. Such approaches use complex 3D rendering rather than simple 2D image transformations and require geometric impostors suitable for visibility determination, especially demanding for dynamic scenes. By separating visibility determination from appearance approximation, we exploit the simplest appearance representation (a 2D image without z) and warp (affine transformation), without sacrificing correct visibility results.

In our approach, the content author identifies geometry that forms the lowest level layers, called *parts*. Parts (*e.g.*, a tree, car, space vehicle, or joint in an articulated figure) contain many polygons and form a perceptual object or object component that is expected to have coherent motion. Very large continuous objects, like terrain, are *a priori* split into component objects. At runtime, for every frame, the visibility relations between parts are incrementally analyzed to generate a sorted list of layers, each containing one or more parts, and an image compositing expression ([Porter84]) on these layers that produces the final image. We assume the renderer can correctly produce hidden-surface-eliminated images for each layer when necessary, regardless of whether the layer contains one or more parts.

Once defined, our approach never splits parts at run-time as in BSP-tree or octree visibility algorithms; the parts are rendered alone or in groups. There are two reasons for this. First, real-time software visibility sorting is practical for hundreds of parts but not for the millions of polygons they

Address: 1 Microsoft Way, Redmond WA 98052.
Email: johnsny@microsoft.com, jedl@microsoft.com

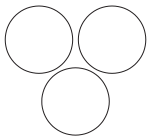


Figure 1: Sorting without splitting: This configuration can be visibility sorted for any viewpoint even though no non-splitting partitioning plane exists.

contain. Second, splitting is undesirable and often unnecessary. In dynamic situations, the number of splits and their location in image space varies as the corresponding split object or other objects in its environment move. Not only is this a major computational burden, but it also destroys coherence, thereby reducing the reuse rate in image-based rendering acceleration or the compression ratio in a layered image stream.

BSP and octree decompositions require global separating planes which induce unnecessary splitting, even though a global separating plane is not required for a valid visibility ordering (Figure 1). We use pairwise occlusion tests between convex hulls or unions of convex hulls around each part. Such visibility testing is conservative, since it fills holes in objects and counts intersections as occlusions even when the intersection occurs only in the invisible (back-facing) part of one object. This compromise permits fast sorting and, in practice, does not cause undue occlusion cycle growth. Less conservative special cases can also be developed, such as between a sphere/cylinder joint (see Appendix B). Our algorithm always finds a correct visibility sort if it exists, with respect to a pairwise occlusion test, or aggregates mutual occluders and sorts among the resulting groups. Moreover, we show that splitting is unnecessary even in the presence of occlusion cycles having no mutually occluding pairs (*i.e.*, no binary occlusion cycles).

The main contribution of this work is the identification of a new and useful problem in computer graphics, that of visibility sorting and occlusion cycle detection on dynamic, multi-polygon objects without splitting, and the description of a fast algorithm for its solution. We introduce the notion of an occlusion graph, which defines the "layerability" criterion using pairwise occlusion relations without introducing unnecessary global partitioning planes. We present a fast method for occlusion culling, and a hybrid incremental algorithm for performing occlusion testing on convex bounding polyhedra. We show how non-binary occlusion cycles can be dynamically handled without grouping the participating objects, by compiling and evaluating an appropriate image compositing expression. We also show how binary occlusion cycles can be eliminated by pre-splitting geometry. Finally, we demonstrate the practicality of these ideas in several situations and applications. Visibility sorting of collections of several hundred parts can be computed at more than 60Hz on a PC.

2 Previous Work

The problem of visibility has many guises. Recent work has considered invisibility culling [Greene93,Zhang97], analytic hidden surface removal [McKenna87,Mulmuley89,Naylor92], and global visibility [Teller93,Durand97]. The problem we solve, production of a layered decomposition that yields the hidden-surface eliminated result, is an old problem in computer graphics of particular importance before hardware z-buffers became widely available [Schumacker69,Newell72,Sutherland74, Fuchs80]. In our approach, we do not wish to eliminate occluded surfaces, but to find the correct layering order, since occluded surfaces in the current frame might be revealed in the next. Unlike the early work, we handle dynamic, multi-polygon objects without splitting; we call this variant of the visibility problem *non-splitting layered decomposition*.

Much previous work in visibility focuses on walkthroughs of static scenes, but a few do consider dynamic situations. [Sudarsky96] uses octrees for the invisibility culling problem, while [Torres90] uses dynamic BSP trees to compute a visibility ordering on all polygons in the scene. Neither technique treats the non-splitting layered decomposition problem, and the algorithms of [Torres90] remain impractical for real-time animations. Visibility algorithms can not simply segregate the dynamic and static elements of the scene and process them independently. A dynamic object can form an occlusion cycle with static objects that were formerly orderable. Our algorithms detect such situations without expending much

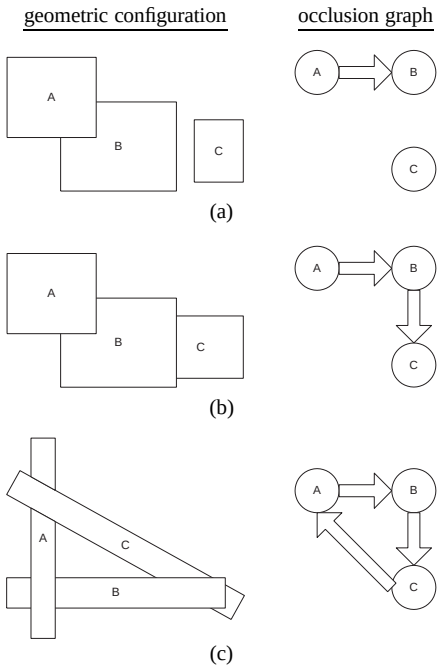


Figure 2: Occlusion graphs: The figure illustrates the occlusion graphs for some simple configurations. (a) and (b) are acyclic, while (c) contains a cycle.

computation on static components.

To accelerate occlusion testing, we use a spatial hierarchy (kd-tree) to organize parts. Such structures (octrees, bintrees, kd-trees, and BSP trees) are a staple of computer graphics algorithms [Fuchs80,Teller91,Funkhouser92, Naylor92,Greene93,Sudarsky96,Shade96]. Our approach generalizes octrees [Greene93,Sudarsky96] and 3D kd-trees [Teller91,Funkhouser92, Shade96] in that it allows a fixed but arbitrarily chosen number of directions in both object and image space. This allows maximum flexibility to tightly bound scenes with a few directions. Our hierarchy is also dynamic, allowing fast rebalancing, insertion, and deletion of objects.

Collision and occlusion detection are similar. We use convex bounding volumes to accelerate occlusion testing, as in [Baraff90,Cohen95, Ponamgi97], and track extents with vertex descent on the convex polyhedra, as in [Cohen95] (although this technique is generalized to angular, or image space, extent tracking as well as spatial). Still, occlusion detection has several peculiarities, among them that an object A can occlude B even if they are nonintersecting, or in fact, very far apart. For this reason, the sweep and prune technique of [Cohen95,Ponamgi97] is inapplicable to occlusion detection. We instead use kd-trees that allow dynamic deactivation of objects as the visibility sort proceeds. Pairwise collision of convex bodies can be applied to occlusion detection; we hybridize techniques from [Chung96a,Chung96b,Gilbert88].

The work of [Max85] deserves special mention as an early example of applying visibility sorting and image compositing to special effects generation. Our work develops the required sorting theory and algorithms.

3 Occlusion Graphs

The central notion for our visibility sorting algorithms is the pairwise occlusion relation. We use the notation $A \rightarrow_E B$ meaning object A occludes object B with respect to eye point E. Mathematically, this relation signifies that there exists a ray emanating from E such that the ray intersects A and then B.[1] It is useful notationally to make the dependence on the eye point implicit so that $A \rightarrow B$ means that A occludes B with respect to an implicit eye point. The arrow "points" to the object that is occluded.

The set of occlusion relations between pairs of the n parts comprising the entire scene forms a directed graph, called the occlusion graph. This notion of occlusion graph is very similar to the priority graph of

[1] A definition for $A \rightarrow_E B$ more suitable for closed objects but harder to compute is that a ray emanating from E hits a front face of A followed by a front face of B.

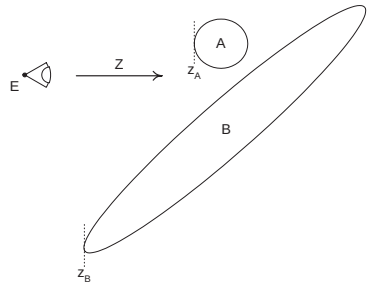


Figure 3: Depth ordering does not indicate visibility ordering: While the minimum depth of object B is smaller than A's ($z_B < z_A$), A occludes B as seen from eye point E. Similarly, by placing E on the right side of the diagram, it can be seen that maximum depth ordering also fails to correspond to visibility ordering.

[Schumacker69] but uses actual occlusion of the objects rather than the results of plane equation tests for pairwise separating planes chosen *a priori* (view independently). Figure 2 illustrates some example occlusion graphs. When the directed occlusion graph is acyclic, visibility sorting is equivalent to topological sorting of the occlusion graph, and produces a (front-to-back) ordering of the objects $\langle O_1, O_2, \ldots, O_n \rangle$ such that $i < j$ implies $O_j \not\to O_i$. Objects so ordered can thus be rendered with correct hidden surface elimination simply by using "Painter's algorithm"; *i.e.*, by rendering O_n, followed by O_{n-1}, and so on until O_1. Thus the final image, I, can be constructed by a sequence of "over" operations on the image layers of each of the objects:

$$I \equiv I_1 \text{ over } I_2 \text{ over } \ldots \text{ over } I_n \qquad (1)$$

where I_i is the *shaped image* of O_i, containing both color and coverage/transparency information [Porter84].

Cycles in the occlusion graph mean that no visibility ordering exists (see Figure 2c). In this case, parts in the cycle are grouped together and analyzed further to generate an image compositing expression (Section 5). The resulting image for the cycle can then be composited in the chain of over operators as above.

This notion of occlusion ignores the viewing direction; only the eye point matters. By taking account of visibility relationships all around the eye, the algorithm described here can respond to rapid shifts in view direction common in interactive settings and critical in VR applications [Regan94].

4 Incremental Visibility Sorting

Our algorithm for incremental visibility sorting and occlusion cycle detection (IVS) is related to the Newell, Newell, and Sancha (NNS) algorithm for visibility ordering a set of polygons [Newell72,Sutherland74]. In brief, NNS sorts a set of polygons by furthest depth and tests whether the resulting order is actually a visibility ordering. NNS traverses the depth-sorted list of polygons; if the next polygon does not overlap in depth with the remaining polygons in the list, the polygon can be removed and placed in the ordered output. Otherwise, NNS examines the collection of polygons that overlap in depth using a series of occlusion tests of increasing complexity. If the polygon is not occluded by any of these overlapping polygons, it can be sent to the output; otherwise, it is marked and reinserted behind the overlapping polygons. When NNS encounters a marked polygon, a cyclic occlusion is indicated and NNS splits the polygon to remove the cycle.

IVS differs from NNS in that it orders aggregate geometry composed of many polygons rather than individual polygons, and identifies and groups occlusion cycles rather than splitting to remove them. Most important, IVS orders incrementally, based on the visibility ordering computed previously, rather than starting from an ordering based on depth.

This fundamental change has both advantages and disadvantages. It is advantageous because depth sorting is an unreliable indicator of visibility order as shown in Figure 3. Applying the NNS algorithm to a coherently changing scene repeatedly computes the same object reorderings (with their attendant costly occlusion tests) to convert the initial depth sort to a visibility sort. The disadvantage is that the sort from the last invocation provides no restriction on the set of objects that can occlude a given object for the current invocation. The NNS depth sort, in contrast, has the useful

IVS(L,G) *[computes visibility sort]*

Input: ordering of non-grouped objects from previous invocation (L)
Output: front-to-back ordering with cyclic elements grouped together (G)
Algorithm:

$G \leftarrow \emptyset$
unmark all elements of L
while L is nonempty
  pop off top(L): A
  if A is unmarked
    if nothing else in L occludes A
      insert A onto G
      unmark everything in L
    else
      [reinsert A onto L]
      mark A
      find element in L occluding A furthest from top(L): F_A
      reinsert A into L after F_A
    endif
  else *[A is marked]*
    form list $S \equiv \langle A, L_1, L_2, \ldots, L_n \rangle$ where $L_1, \ldots, L_n$
      are the largest consecutive sequence of marked elements,
      starting from top(L)
    if detect_cycle(S) then
      [insert cycle as grouped object onto L]
      group cycle-forming elements of S into grouped object C
      delete all members of C from L
      insert C (unmarked) as top(L)
    else
      [reinsert A onto L]
      find element in L occluding A furthest from top(L): F_A
      reinsert A into L after F_A
    endif
  endif
endwhile

Figure 4: IVS algorithm. The top object, A, in the current ordering (list L) is examined for occluders. If nothing occludes A, it is inserted in the output list G. Otherwise, A is marked and reinserted behind the furthest object in the list that occludes it, F_A. When a marked object is encountered, the sequence of consecutively marked objects starting at the top of the list is checked for an occlusion cycle using detect_cycle. If an occlusion cycle is found, the participating objects are grouped and reinserted on top of L. This loop is repeated until L is empty; G then contains the sorted list of parts with mutual occluders grouped together.

detect_cycle(S) *[finds a cycle]*

Input: list of objects $S \equiv \langle S_1, S_2, \ldots, S_n \rangle$
Output: determination of existence of a cycle and a list of
  cycle-forming objects, if a cycle is found.
Algorithm:

if $n \leq 1$ return NO_CYCLE
$i_1 \leftarrow 1$
for $j = 2$ to $n + 1$
  if S_{i_k} occludes $S_{i_{j-1}}$ for $k < j - 1$ then
    cycle is $\langle S_{i_k}, S_{i_{k+1}}, \ldots, S_{i_{j-1}} \rangle$
    return CYCLE
  else if no occluder of $S_{i_{j-1}}$ exists in S then
    return NO_CYCLE
  else
    let S_k be an occluder of $S_{i_{j-1}}$
    $i_j \leftarrow k$
  endif
endfor

Figure 5: Cycle detection algorithm used in IVS. This algorithm will find a cycle if any initial contiguous subsequence of $1 < m \leq n$ vertices $\langle S_1, S_2, \ldots, S_m \rangle$ forms a *cyclically-connected subgraph*; *i.e.*, a subgraph in which every part is occluded by at least one other member of the subgraph. For subgraphs which are not cyclically connected, the algorithm can fail to find existing cycles, but this is not necessary for the correctness of IVS (for example, consider the occlusion graph with three nodes A, B, and C where $A \to B \to A$ and initial list $\langle C, A, B \rangle$).

L	G	comment
ABC	$\emptyset$	initial state
BC	A	insert A onto G
C	AB	insert B onto G
$\emptyset$	ABC	insert C onto G

Figure 6: IVS Example 1: Each line shows the state of L and G after the next while loop iteration, using the graph of Figure 2(b) and initial ordering ABC.

L	G	comment
CBA	$\emptyset$	initial state
BC^*A	$\emptyset$	mark C and reinsert after B
C^*AB^*	$\emptyset$	mark B and reinsert after A
AB^*C^*	$\emptyset$	A unmarked, so reinsert C
BC	A	insert A onto G, unmark everything
C	AB	insert B onto G
$\emptyset$	ABC	insert C onto G

Figure 7: IVS Example 2: Using the graph of Figure 2(b), this time with initial ordering CBA. The notation P^* is used to signify marking. The step from 3 to 4 reinserts C into L because there is an unmarked element, A, between C and the furthest element occluding it, B.

L	G	comment
ABC	$\emptyset$	initial state
BCA^*	$\emptyset$	mark A and reinsert
CA^*B^*	$\emptyset$	mark B and reinsert
$A^*B^*C^*$	$\emptyset$	mark C and reinsert
(ABC)	$\emptyset$	group cycle
$\emptyset$	(ABC)	insert (ABC) onto G

Figure 8: IVS Example 3: Using the graph of Figure 2(c) with initial ordering ABC. The notation $(P_1, P_2, \ldots, P_r)$ denotes grouping.

property that an object Q further in the list from a given object P, and all objects after Q, can not occlude P if Q's min depth exceeds the max depth of P. Naively, IVS requires testing potentially all n objects to see if any occlude a given one, resulting in an $O(n^2)$ algorithm. Fortunately, we will see in the next section how the occlusion culling may be sped up using simple hierarchical techniques, actually improving upon NNS occlusion culling (see Section 8).

The IVS algorithm is presented in Figures 4 and 5. Mathematically, the IVS algorithm computes an incremental topological sort on the strongly connected components of the directed occlusion graph (see [Sedgewick83] for background on directed graphs, strongly connected components, and topological sort). A strongly connected component (SCC) in the occlusion graph is a set of mutually occluding objects, in that for any object pair A and B in the SCC, either $A \rightarrow B$ or there exist objects, $X_1, X_2, \ldots, X_s$ also in the SCC such that

$$A \rightarrow X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_s \rightarrow B.$$

The IVS algorithm finds the parts comprising each SCC, and optionally computes the occlusion subgraph of the members of each SCC to resolve nonbinary cycles without aggregating layers (Section 5).

A series of example invocations of the IVS algorithm for some of the occlusion graphs in Figure 2 are presented in Figures 6, 7, and 8. A proof of correctness is contained in [Snyder97].

The IVS algorithm takes advantage of coherence in the visibility ordering from the previous frame. When a given object A is popped off the list, it is likely that few objects further in the list will occlude it. Typically, no objects will be found to occlude A and it will be immediately inserted onto G. If we can quickly determine that no objects occlude A, and the new ordering requires no rearrangements, the algorithm verifies that the new order is identical to the old with computation $O(n \log n)$ in the total number of objects. In essence, the algorithm's incrementality allows it to examine only a small subset of the potentially $O(n^2)$ arcs in the occlusion graph.

We assume that occlusion cycles (SCCs) will be small and of limited duration in typical scenes. This assumption is important since the cycle detection algorithm has quadratic complexity in the number of cycle elements. The visibility sorting algorithm does not attempt to exploit coherence in persistent occlusion cycles.

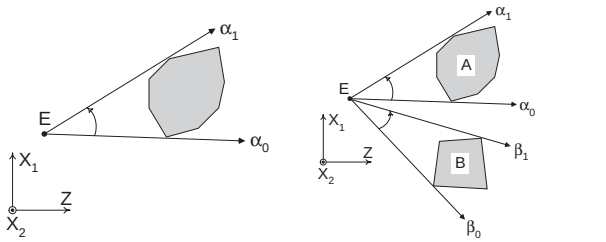


Figure 9: Angular extent occlusion culling: Angular extents are defined with respect to an eye point E and a orthogonal coordinate frame (X_1, X_2, Z) where X_2 (out of the page) is perpendicular to the plane in which angles are measured, Z defines the zero angle, and X_1 defines an angle of $+\pi/2$ radians. The resulting extent is simply an interval: $[\alpha_0, \alpha_1]$. To determine that $B \not\rightarrow A$ (right side of figure), we test for empty interval intersection.

As the number of re-arrangements required in the new order increases (*i.e.*, as the coherence of the ordering decreases) the IVS algorithm slows down, until a worst case scenario of starting from what is now a completely reversed ordering requires $O(n^2)$ outer while loop iterations. This is analogous to using insertion sort for repeatedly sorting a coherently changing list: typically, the sort is $O(n)$, but can be $O(n^2)$ in pathologically incoherent situations.

The algorithm's complexity is bounded by $(n+r)(s+co+c^2)$ where r is the number of reinsertions required, c is the maximum number of objects involved in an occlusion cycle, o is the maximum number of primitive occluders of a (possibly grouped) object, and s is the complexity of the search for occluders of a given object. The first factor represents the number of outer while-loop iterations of IVS. In the second factor, the three terms represent time to find potential occluders, to reduce this set to actual occluders (see Section 4.1.3), and to detect occlusion cycles. Typically, $r \sim O(n)$, $c \sim O(1)$, $o \sim O(1)$, and $s \sim O(\log n)$ resulting in an $O(n \log n)$ algorithm. In the worst case, many reinsertions are required, many objects are involved in occlusion cycles, and many objects occlude any given object so that $r \sim O(n^2)$, $c \sim O(n)$, $o \sim O(n)$, and $s \sim O(n)$ resulting in an $O(n^4)$ algorithm. This analysis assumes that occlusion detection between a pair of parts requires constant time.

When the animation is started and at major changes of scene, there is no previous visibility sort to be exploited. In this case, we use an initial sort by distance from the eye point to the centroid of each part's bounding hull. Using a sort by z is less effective because it sorts objects behind the eye in reverse order; sorting by distance is effective even if the view direction swings around rapidly.

4.1 Occlusion Culling

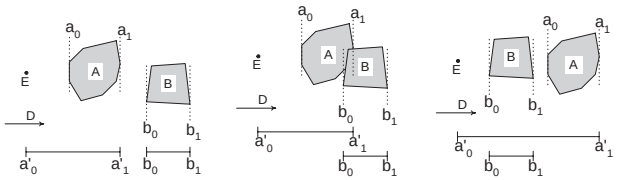
The fundamental query of the IVS algorithm determines which current objects occlude a given (possibly grouped) object. To quickly cull the list of candidate occluders to as small a set as possible, we bound each part with a convex polyhedron and determine the spatial extent of this convex bound with respect to a predetermined set of directions, as in [Kay86]. These directions are of two types. *Spatial extents* are projections along a given 3D vector. *Angular extents* are projected angles with respect to a given eye point and axis. Spatial extents are defined by extremizing (maximizing and minimizing) $\mathcal{S}(P) \equiv D \cdot P$ over all points P in the convex hull. Angular extents are defined similarly by extremizing[2]

$$\mathcal{A}(P) \equiv \tan^{-1}\left(\frac{(P-E)\cdot Z}{(P-E)\cdot X_1}\right) \quad (2)$$

where E is the "eye" point, Z defines the zero angle direction, and X_1 defines the positive angles.

Given two objects, A and B, with interval bounds for each of their extents, occlusion relationships can be tested with simple interval intersection tests performed independently for each extent, as shown in Figures 9 and 10. The content author chooses the number of spatial (k_s) and angular (k_a) extents and their directions; let $k \equiv k_s + k_a$ be the total number. If any of

[2]Care must be taken when the denominator is close to 0. This is easily accomplished in the C math library by using the function `atan2`.



(a) $B \not\rightarrow A$ (b) $B \rightarrow A$ (c) $B \rightarrow A$

Figure 10: Spatial extent occlusion culling: Spatial extents are defined with respect to a direction D. To test whether $B \rightarrow A$, A's spatial extent $[a_0, a_1]$ is expanded by $E \cdot D$ to yield $[a'_0, a'_1]$. Three cases can occur. In (a), $[a'_0, a'_1]$ is disjoint from B's extent $[b_0, b_1]$, so $B \not\rightarrow A$. In (b), $[a'_0, a'_1]$ overlaps with $[b_0, b_1]$, so $B \rightarrow A$ is possible. In (c), $[a'_0, a'_1]$ overlaps with $[b_0, b_1]$ even though A's extent $[a_0, a_1]$ is disjoint from B. Again, $B \rightarrow A$ is possible. Note that in case (b) and (c), the occlusion cull tests must determine $B \rightarrow A$ for all k extents before concluding B is a possible occluder of A.

the k tests finds that $B \not\rightarrow A$ then the test can be concluded and B rejected as an occluder without testing more extents.

Note that the algorithm computes all intersecting pairs, which is a useful computational by-product for simulation. View frustum culling is made trivial by computing the angular extents of the visible region once at the start of each frame and determining whether each objects' angular extents intersect it.

4.1.1 Tracking Extents on Convex Hulls

Spatial extent directions can be fixed in space (*e.g.*, the coordinate axes, but note that arbitrary directions are allowed) or tied to the camera. Camera-independent spatial extents only need to be updated when the object moves; camera-dependent spatial extents must be updated when the object or the camera moves. Angular extents must also be updated whenever the object or camera moves. For the results in Section 8, in one case (Tumbling Toothpicks) we used two orthogonal angular extents (screen x and y directions) and the orthogonal camera-dependent spatial extent (Z). In another case with many unmoving objects (Canyon Flyby), we used 3 mutually orthogonal camera-independent spatial extents.

For convex bounding polyhedra, spatial and angular extents can be updated simply by "sliding downhill" (*i.e.*, gradient descent) from vertex to neighboring vertex, evaluating the objective function ($\mathcal{S}$ or $\mathcal{A}$) at each vertex. At each iteration, the neighboring vertex having the minimum value is accepted as the starting point for the next iteration. If no neighbors have a smaller objective function, then the computation is halted with the current vertex returned as the minimizer. Small motions of the convex hull or the spatial/angular reference frame move the new extremal vertex at most a few neighbors away from the last one. By starting with the extremal vertex from the last query, coherence in object and camera motions is thus exploited.

4.1.2 Accelerating Occlusion Queries with Kd-Trees

We have reduced the problem of finding all potential occluders of an object A to

1. forming a query extent for A, in which an k-dimensional interval is created by taking the angular extents without change and the spatial extents after enlarging by $E \cdot D$, and
2. finding all objects that overlap this query.

We hierarchically organize part extents using a kd-tree to accelerate finding the set of overlapping extents for a given query.

A kd-tree [Bentley75] is a binary tree which subdivides along k fixed dimensions. Each node T in the tree stores both the dimension subdivided ($T.i$) and the location of the partitioning point ($T.v$). Object extents whose $T.i$-th dimension interval lower bound is less than $T.v$ are placed in the left child of node T; those whose upper bound is greater than $T.v$ are placed in the right child. Objects which straddle kd-planes are simply inserted into both subtrees. Note that the planes are not used directly to determine the visibility order; the structure simply accelerates occlusion queries.

A simple minimum cost metric is used to determine a subdivision point for a list of intervals, representing the 1D extents of the set of objects with respect to one of the k_a angular or k_s spatial directions. Our cost metric sums the length of the longer of the left and right sublists and the number of intervals shared between left and right. Avoiding lopsided trees and trees in which many objects are repeated in both subtrees is desirable since such trees tend to degrade query performance in the average case. The cost can be computed with a simple traversal of a sorted list containing both upper and lower bounds; details can be found in [Snyder97].

To build the kd-tree, we begin by sorting each of the k interval sets to produce *k 1D sorted bound lists*, containing both upper and lower bounds. The kd-tree is then built recursively in a top-down fashion. To subdivide a node, the partitioning cost is computed for each of the k bound lists, and the dimension of lowest cost actually used to partition. Bound lists for the partitioned children are built in sorted order by traversing the sorted parent's list, inserting to either or both child lists according to the computed partitioning. We then recurse to the left and right sides of the kd-tree. The algorithm is terminated when the longer child list is insufficiently smaller than its parent (we use a threshold of 10). A node T in the final kd-tree stores the 1D sorted bound list only for dimension $T.i$, which is used to update the subdivision value $T.v$ in future queries, and to shift objects between left and right subtrees as they move. The other lists are deleted.

Since rebuilding is relatively expensive, the algorithm also incorporates a quick kd-tree rebalancing pass. To rebalance the kd-tree as object extents change, we visit all its nodes depth-first. At each node T, the 1D sorted bound list is re-sorted using insertion sort and the cost algorithm is invoked to find a new optimal subdivision point, $T.v$. Extents are then repartitioned with respect to the new $T.v$, shifting extents between left and right subtrees. Extent addition is done lazily (*i.e.*, only to the immediate child), with further insertion occurring when the child nodes are visited. Extent deletion is done immediately for all subtrees in which the extent appears, an operation that can be done efficiently by recording a (possibly null) left and right child pointer for each extent stored in T. Note that coherent changes to the object extents yield an essentially linear re-sort of bound lists, and few objects that must be shifted between subtrees.

It is important to realize that the coherence of kd-tree rebalancing depends on fixing the subdivision dimension $T.i$ at each node. If changes in the subdivided dimension were allowed, large numbers of extents could be shifted between left and right subtrees, eliminating coherence in all descendants. Fixing $T.i$ but not $T.v$ restores coherence, but since $T.i$ is computed only once, the tree can gradually become less efficient for query acceleration as object extents change. This problem can be dealt with by rebuilding the tree after a specified number of frames or after measures of tree effectiveness (*e.g.*, tree balance) so indicate. A new kd-tree can then be rebuilt as a background process over many frames while simultaneously rebalancing and querying an older version.

Querying the kd-tree involves simple descent guided by the query. At a given node T, if the query's $T.i$-th interval lower bound is less than $T.v$, then the left subtree is recursively visited. Similarly, if the query's $T.i$-th interval upper bound is greater than $T.v$ then the right subtree is recursively visited. When a terminal node is reached, extents stored there are tested for overlap with respect to all k dimensions. Overlapping extents are accumulated into an occluder list. An extent is inserted only once in the occluder list, though it may occur in multiple leaf nodes.

An additional concern is that the occlusion query should return occluders of an object *A that have not already been inserted into the output list.* Restricting the set of occluders to the set remaining in L can be accomplished by activating/deactivating extents in the kd-tree.When A is popped off the list L in the IVS algorithm, all objects grouped within it are deactivated. Deactivated objects are handled by attaching a flag to each object in the list stored at each terminal node of the kd-tree. Deactivating an object involves following its left and right subtree pointers, beginning at the kd root, to arrive at terminal lists containing the object to be deactivated. Activation is done similarly, with the flag set oppositely. Counts of active objects within each kd-tree node are kept so that nodes in which all objects have been deactivated can be ignored during queries.

4.1.3 Avoiding Occlusion Cycle Growth

The occlusion testing described so far is conservative, in the sense that possible occluders of an object can be returned which do not in fact occlude it. There are two sources of this conservativeness. First, occlusion is tested with respect to a fixed set of spatial and/or angular extents, which essentially creates an object larger than the original convex hull and thus more likely to be occluded. Second, extents for grouped objects are computed by simple

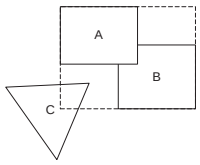


Figure 11: Occlusion cycle growth with grouped objects: In this example, A and B have been grouped because they are mutually occluding. A simple bound around their union, shown by the dashed lines, is occluded by object C, even though the objects themselves are not. We therefore use the bounded extents around grouped objects for a quick cull of nonoccluders, but further test objects which are not so culled to make sure they occlude at least one primitive element of the grouped object.

unioning of the extents of the members, even though the unioned bound may contain much empty space, as shown in Figure 11. The next section will show how to compute an exact occlusion test between a pair of convex objects, thus handling the first problem. This section describes a more stringent test for grouped objects which removes the second problem.

Occlusion testing that is too conservative can lead very large groupings of objects in occlusion cycles. In the extreme case every object is inserted into a single SCC. This is especially problematic because of the second source of conservatism – that bounds essentially grow to encompass all members of the current SCC, which in turn occlude further objects, and so on, until the SCC becomes very large.

To handle this problem, we perform additional tests when a grouped object A is tested for occlusion. A's unioned extents are used to return a candidate list of possible occluders, as usual. Then the list of occluders is scanned to make sure each occludes at least one of the primitive members of A, using a simple k-dimensional interval intersection test. Any elements of the list that do not occlude at least one member of A are rejected, thus ensuring that "holes" within the grouped object can be seen through without causing occlusions. Finally, remaining objects can be tested against primitive members of A using the exact occlusion test.

4.2 Occlusion Testing

The algorithms in Section 4.1 provide a fast but conservative pruning of the set of objects that can possibly occlude a given object A. To produce the set of objects that actually occlude A with respect to the convex hull bounds, we apply an exact test of occlusion for primitive object pairs (A, B), which determines whether $B \rightarrow A$. The test is used in the IVS algorithm by scanning the list of primitive elements of the possibly grouped object A and ensuring that at least one occluder in the returned list occludes it, with respect to the exact test. The exact test is thus used as a last resort when the faster methods fail to reject occluders.

The exact occlusion test algorithm is as follows:

```
ExactConvexOcclusion(A,B,E)      [returns whether B →_E A]

    if all (non eye-expanded) spatial extents of A and B intersect
        initiate 3D collision tracking for ⟨A, B⟩, if not already
        if A and B collide, return B → A
        if E on same side of separating plane as A, return B ↛ A
    endif

    if B contains eye point E, return B → A     [B occludes everything]
    initiate occlusion tracking for ⟨B, A⟩, if not already
    return result of occlusion test
```

Both the collision and occlusion query used in the above algorithm can be computed using the algorithm in Appendix A. While the collision query is not strictly necessary, it is more efficient in the case of a pair of colliding objects to track the colliding pair once rather than tracking two queries which bundle the eye point with each of the respective objects. For scenes in which collisions are rare, the direct occlusion test should be used.

The IVS algorithm is extended to make use of a hash table of object pairs for which 3D collision or occlusion tracking have been initialized, allowing fast access to the information. Tracking is discontinued for a pair if the information is not accessed after more than one frame.

Note that further occlusion resolution is also possible with respect to the actual objects rather than convex bounds around them. It is also possible to inject special knowledge in the occlusion resolution process, such as the fact that a given separating plane is known to exist between certain pairs of objects, like joints in an articulated character or adjacent cells in a pre-partitioned terrain. Special purpose pairwise visibility codes can also be developed; Appendix B provides an example for a cylinder with endcap tangent to a sphere that provides a visibility heuristic for articulated joints in animal-like creatures.

4.3 Conditioning Sort

After each IVS invocation, we have found it useful to perform a conditioning sort on the output that "bubbles up" SCCs based on their midpoint with respect to a given extent. More precisely, we reorder according to the absolute value of the difference of the midpoint and the projection of the eye point along the spatial extents. The camera-dependent Z direction is typically used as the ordering extent, but other choices also provide benefit. An SCC is only moved up in the order if doing so does not violate the visibility ordering; *i.e.*, the object does not occlude the object before which it is inserted. This conditioning sort smooths out computation over many queries. Without it, unoccluding objects near the eye can remain well back in the ordering until they finally occlude something, when they must be moved in front of many objects in the order, reducing coherence. The conditioning sort also sorts parts within SCCs according to extent midpoint, but ignoring occlusion relationships (since the SCC is not visibility sortable).

5 Resolving Non-Binary Cycles

Following [Porter84], we represent a shaped image as a 2D array of 4-tuples, written

$$A \equiv [A_r, A_g, A_b, A_\alpha]$$

where A_r, A_g, A_b are the color components of the image and A_α is the transparency, in the range $[0, 1]$.

Consider the cyclic occlusion graph and geometric situation shown in Figure 2c. Clearly,

$$A \text{ over } B \text{ over } C$$

produces an incorrect image because $C \rightarrow A$ but no part of C comes before A in the ordering. A simple modification though produces the correct answer:

$$A \text{ out } C + B \text{ out } A + C \text{ out } B.$$

where "out" is defined as

$$A \text{ out } B \equiv A(1 - B_\alpha).$$

This expression follows from the idea that objects should be attenuated by the images of all occluding objects. Another correct expression is

$$(C \text{ atop } A) \text{ over } B \text{ over } C$$

where "atop" is defined as

$$A \text{ atop } B \equiv AB_\alpha + (1 - A_\alpha)B.$$

In either case, the expression correctly overlays the relevant parts of occluding objects over the occluded objects, using only shaped images for the individual objects (refer to Figure 12). Technically, the result is not correct at any pixels partially covered by all three objects, since the matte channel encodes coverage as well as transparency. Such pixels tend to be isolated, if they exist, and the resulting errors of little significance.[3]

The above example can be generalized to any collection of objects with a known occlusion graph having no *binary cycles*: cycles of the form $A \rightarrow B, B \rightarrow A$. The reason binary cycles can not be handled is that in the region of intersection of the bounding hulls of A and B, we simply have no information about which object occludes which. Note also that the compositing expression in this case reduces to $A \text{ out } B + B \text{ out } A$ which incorrectly eliminates the part of the image where $A \bigcap B$ projects.

A correct compositing expression for n shaped images I_i is given by

$$\sum_{i=1}^{n} I_i \underset{\{j \mid O_j \rightarrow O_i\}}{\text{OUT}} I_j \tag{3}$$

The notation OUT with a set subscript is analogous to the multiplication accumulator operator Π, creating a chain of "out" operations, as in

$$D \underset{\{A,B,C\}}{\text{OUT}} = D \text{ out } A \text{ out } B \text{ out } C.$$

[3] Recall too that overlaying shaped images where the matte channel encodes coverage is itself an approximation since it assumes uncorrelated silhouette edges.

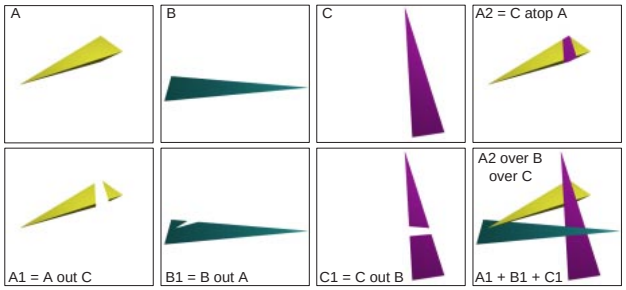


Figure 12: Compositing expressions for cycle breaking: The original sprite images are shown as A, B, C. Using "over-atop", the final image is formed by (*C* atop *A*) over *B* over *C*. Using "sum-of-outs", the final image is formed by (*A* out *C*) + (*B* out *A*) + (*C* out *B*).

In words, (3) sums the image for each object O_i, attenuated by the "out" chain of products for each object O_j that occludes O_i (Figure 12, bottom row).

An alternate recursive formulation using atop is harder to compile but generates simpler expressions. As before, we have a set of objects $O = \{O_i\}$ together with an occlusion graph G for O containing no binary cycles. The subgraph of G induced by an object subset $X \subset O$ is written G_X. Then for any $O_* \in O$

$$I(G_O) = \left(I(G_{\{O_i|O_i \to O_*\}}) \text{ atop } I(O_*)\right) \text{ over } I(G_{O-\{O_*\}}) \qquad (4)$$

where $I(G)$ represents the shaped image of the collection of objects using the occlusion graph G. In other words, to render the scene, we can pick any isolated object O_*, find the expression for the subgraph induced by those objects occluding O_*, and compute that expression "atop" O_* (Figure 12, top right). That result is then placed "over" the expression for the subgraph induced by removing O_* from the set of objects O. Note also that the above expression assumes $I(G_\emptyset) = 0$.

Proofs of correctness of the two expressions is available in a technical report [Snyder98].

Compositing Expression Compilation

An efficient approach to generating an image compositing expression for the scene uses the IVS algorithm to produce a visibility sorted list of SCCs. Thus the images for each SCC can be combined using a simple sequence of "over" operations as in Expression (1). Most SCCs are *singletons* (containing a single object). Non-singleton SCCs are further processed to merge binary cycles, using the occlusion subgraph of the parts comprising the SCC. Merging must take place iteratively in case binary cycles are present between objects that were merged in a previous step, until there are no binary cycles between merged objects. We call such merged groups BMCs, for *binary merged components*. Expression (3) or (4) is then evaluated using the resulting merged occlusion graph to produce an expression for the SCC. Each BMC must be grouped into a single layer, but not the entire SCC. For example, Figure 2(c) involves one SCC but three BMCs, since there are no binary cycles.

It is clear that Expression (3) can be evaluated using two image registers: one for accumulating a series of "out" operations for all image occluders of a given object, and another for summing the results. Expression (4) can be similarly compiled into an expression using two image resisters: one for "in" or "out" operations and one for sum accumulation [Snyder98]. Two image registers thus suffice to produce the image result for any SCC. An efficient evaluation for the scene's image requires a third register to accumulate the results of the "over" operator on the sorted sequence of SCCs. This third register allows segregation of the SCCs into separately compilable units.

Given such a three-register implementation, it can be seen why Expression (4) is more efficient. For example, for a simple ring cycle of n objects; *i.e.*, a graph

$$O_1 \to O_2 \to \cdots \to O_n \to O_1$$

the "sum-of-outs" formulation (Expression 3) produces

$$I(O_1)\text{out}I(O_n) + I(O_2)\text{out}I(O_1) + I(O_3)\text{out}I(O_2) + \cdots + I(O_n)\text{out}I(O_{n-1})$$

with n "out" and $n-1$ addition operations, while the "over-atop" formulation (Expression 4) produces

$$(I(O_n) \text{ in } I(O_1) + I(O_1) \text{ out } I(O_n)) \text{ over } I(O_2) \text{ over } \cdots \text{ over } I(O_n)$$

with $n-1$ "over", 1 "in", 1 "out", and 1 addition operators. Assuming "over" is an indispensable operator for hardware implementations and is thus atomic, the second formulation takes advantage of "over" to reduce the expression complexity.

6 Pre-Splitting to Remove Binary Cycles

The use of convex bounding hulls in occlusion testing is sometimes overly conservative. For example, consider a pencil in a cup or an aircraft flying within a narrow valley. If the cup or valley form a single part, our visibility sorting algorithm will always group the pencil and cup, and the aircraft and valley, in a single layer (BMC) because their convex hulls intersect. In fact, in the case of the valley it is likely that nearly all of the scene's geometry will be contained inside the convex hull of the valley, yielding a single layer for the entire scene.

To solve this problem, we pre-split objects that are likely to cause unwanted aggregation of parts. Objects that are very large, like terrain, are obvious candidates. Foreground objects that require large rendering resources and are known to be "containers", like the cup, may also be pre-split. Pre-splitting means replacing an object with a set of parts, called *split parts*, whose convex hull is less likely to intersect other moving parts. With enough splitting, the layer aggregation problem can be sufficiently reduced or eliminated.

Simple methods for splitting usually suffice. Terrain height fields can be split using a 2D grid of splitting planes, while rotationally symmetric containers, like a cup, can be split using a cylindrical grid. A 3D grid of splitting planes can be used for objects without obvious projection planes or symmetry (*e.g.*, trees). On the other hand, less naive methods that split more in less convex regions can reduce the number of split parts, improving performance. Such methods remain to be investigated in future work.

Pre-splitting produces a problem however. At the seam between split neighbors the compositor produces a pixel-wide gap, because its assumption of uncorrelated edges is incorrect. The split geometry exactly tessellates any split surfaces; thus alpha (coverage) values should be added at the seam, not over'ed. The result is that seams become visible.

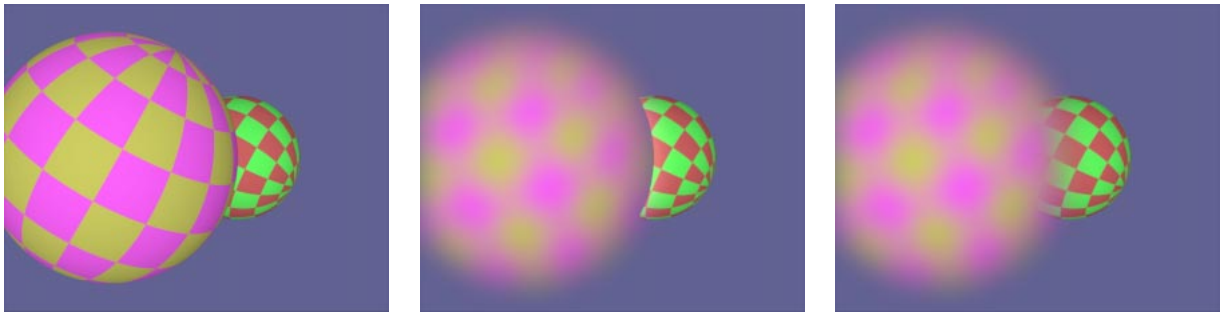
To solve this problem, we extend the region which is included in each split object to produce overlapping split parts, a technique also used in [Shade96]. While this eliminates the visible seam artifact, it causes split parts to intersect, and the layer aggregation problem recurs. Fortunately, adjacent split parts contain the same geometry in their region of overlap. We therefore add pairwise separating planes between neighbors, because both agree on the appearance within the region of overlap so either may be drawn. This breaks the mutual occlusion relationship between neighbors and avoids catastrophic layer growth. But we use the convex hulls around the "inflated" split parts for testing with all other objects, so that the correct occlusion relationship is still computed.

Note that the occlusion sort does not preclude splitting arrangements like hexagonal terrain cells that permit no global partitioning planes. All that is required is pairwise separation.

7 Visibility Correct Depth of Field

2D image blurring is a fast method for simulating depth of field effects amenable to hardware implementation [Rokita93]. Unfortunately, as observed in [Cook84], any approximation that uses a single hidden-surface-eliminated image, including [Potmesil81,Rokita93], causes artifacts because no information is available for occluded surfaces made visible by depth of field. The worst case is when a blurry foreground object occludes a background object in focus (Figure 13). As shown in the figure, the approximation of [Potmesil81] sharpens the edge between the foreground and background objects, greatly reducing the illusion. Following [Max85], but using correct visibility sorting, we take advantage of the information in layer images that would ordinarily be eliminated to correct these problems.

The individual image layers are still approximated by spatially invariant blurring in the case of objects having small depth extent, or by the spatially varying convolution from [Potmesil81]. Image compositing is used between layers. Since a substantial portion of depth of field cues come from



(a) no depth of field (b) single layer depth of field approx. (c) two layer visibility compositing approx.

Figure 13: Simulating depth of field with image blurring.

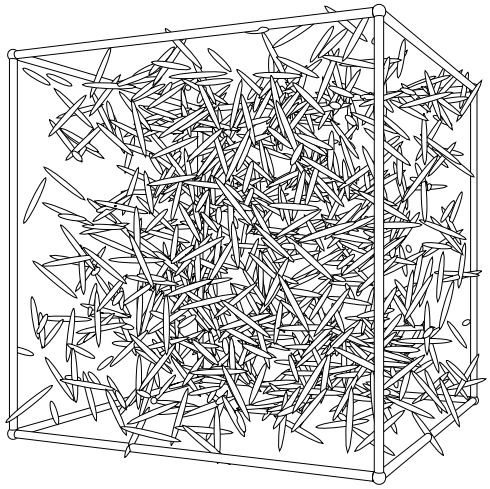


Figure 14: Toothpick example (nobjs=800, uniform scale): This image shows a frame from the first experiment, drawn with hidden line elimination by using Painter's algorithm with the computed visibility order. For the hidden line processing, singleton SCCs are simply drawn by finding the part's silhouette, filling its interior in white and then its boundary as a black polyline. Nonsingleton SCCs are further processed to find visible intersection and silhouette edges dynamically, but only the few objects comprising the SCC need be considered, not the entire scene.

edge relations between foreground and background objects, we consider this a good approximation, although blurring without correctly integrating over the lens only approximates the appearance of individual parts.

Grouping parts in a BMC because of occlusion undecomposability exacts a penalty. Such grouping increases the depth extent of the members of the group so that the constant blur approximation or even the more expensive depth-based convolution incur substantial error. For groupings of large extent, the renderer could resort to rendering integration using the accumulation buffer [Haeberli90]. Such integration requires many rendering passes (23 were used in images from [Haeberli90]), representing a large allocation of system resources to be avoided when simple blurring suffices.

8 Results

All timings are reported for *one processor* of a Gateway E5000-2300MMX PC with dual Pentium II 300MHz processors and 128MB of memory. Measured computation includes visibility sorting and kd-tree building and rebalancing. The kd-tree was built only once at the start of each animation; the amortized cost to build it is included in the "average" cpu times reported.

Tumbling Toothpicks

The first results involve a simulation of tumbling "toothpicks", eccentric ellipsoids, moving in a cubical volume (Figure 14). The toothpicks bounce off the cube sides, but are allowed to pass through each other. Each toothpick contains 328 polygons and forms one part. There are 250 frames in the animation.

In the first series of experiments, we measured cpu time per frame, as a function of number of toothpicks (Figure 15). Time per frame averaged over the entire animation and maximum time for any frame are both reported. One experiment, labeled "us" for *uniform scale* in the figure, adds more toothpicks of the same size to the volume. This biases the occlusion complexity superlinearly with number of objects, since there are many more collisions and the size of the average occlusion cycle increases. With enough toothpicks, the volume becomes filled with a solid mass of moving geometry, forming a single SCC. As previously discussed, the algorithm is designed for situations in which occlusion cycles are relatively small.

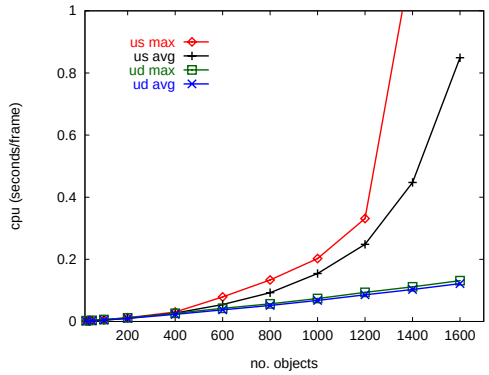
A more suitable measure of the algorithm's complexity preserves the average complexity per unit volume and simply increases the visible volume. This effect can be achieved by scaling the toothpicks by the cube root of their number ratio, so as to preserve average distance between toothpicks as a fraction of their length. The second experiment, labeled "ud" for *uniform density* presents these results. The results demonstrate the expected $O(n \log n)$ rate of growth. The two experiments are normalized so that the simulations are identical within timing noise for nobjs=200: the uniform density experiment applies the scale $(200/\text{nobjs})^{\frac{1}{3}}$ to the toothpicks of the other trials. In particular, note that a simulation with 200 toothpicks (220 total objects including cube parts), can be computed at over 100Hz, making it practical for real-time applications. To verify the above scaling assumptions, the following table summarizes some visibility statistics (averaged over all frames of the animation) for the baseline scene with 200 toothpicks and the two scenes with 1600 toothpicks (uniform density, uniform scale):

measurement	nobjs=200	nobjs=1600 (uniform density)	nobjs=1600 (uniform scale)
fraction of SCCs that are nonsingleton	.0454	.04633	.2542
fraction of parts in nonsingleton SCCs	.0907	0.0929	.5642
average size of nonsingleton SCCs	2.097	2.107	3.798
max size of SCCs	2.64	3.672	45.588

With the exception of the "max size of nonsingleton SCC" which we would expect to increase given that the 1600 object simulation produces greater probability that bigger SCCs will develop, the first two columns in the table are comparable, indicating a reasonable scaling, while the third indicates much greater complexity. Note also that the large maximum cpu time for the 1400 and 1600 uniform scale trials is due to the brief existence of much larger than average sized occlusion cycles.

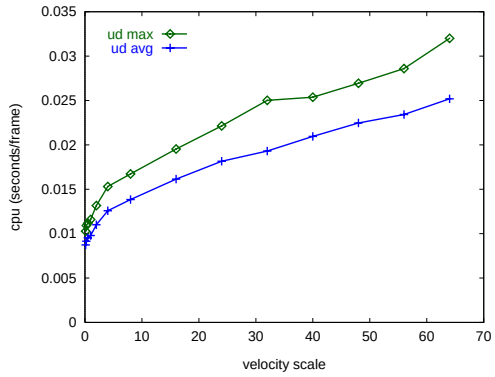
The second experiment measures cpu time with varying coherence. We globally scale the rate of camera movement and the linear and angular velocities of the toothpicks (Figure 16). The number of toothpicks was fixed at 200; the trial with velocity scale of 1 is thus identical to the trial with nobjs=200 in Figure 15. The algorithm is clearly sensitive to changing coherence, but exhibits only slow growth as the velocities become very large. Not surprisingly, the difference between average and worst case query times increases as coherence decreases, but the percentage difference remains fairly constant, between 17% and 30%.

To calibrate the results of the second experiment, let S be the length of the image window and W the length of the cube side containing the



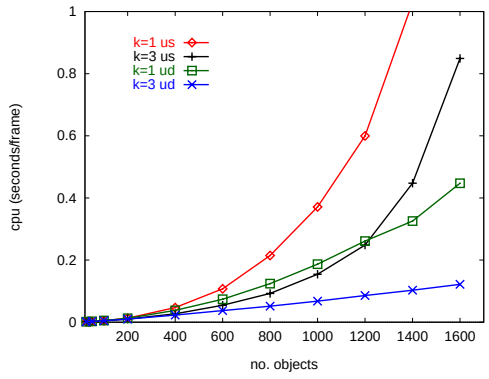
no. objs	25	50	100	200	400	800	1600
avg. cpu (ms) [ud]	1.51	2.51	4.81	9.97	23.1	51.7	122
max cpu (ms) [ud]	2.40	3.55	6.28	11.8	26.3	56.9	131
avg. cpu (ms) [us]	1.32	2.17	4.21	10.1	27.0	92.5	849
max cpu (ms) [us]	2.28	3.11	5.57	11.9	30.5	134	3090

Figure 15: Performance with increasing number of objects.



vel. scl.	.25	.5	1	2	4	8	16	32	64
avg cpu (ms)	9.15	9.52	9.78	11.0	12.6	13.8	16.1	19.3	25.2
max cpu (ms)	10.9	11.2	11.6	13.2	15.3	16.7	19.5	25.0	32.0

Figure 16: Performance with increasing velocity (decreasing coherence).



no. objs	50	100	200	400	800	1600
k = 3 cpu (ms) [ud]	2.51	4.81	9.97	23.1	51.7	122
k = 1 cpu (ms) [ud]	2.65	5.48	12.7	38.0	124	447
% diff. [ud]	5.55%	14.0%	27.7%	64.6%	140%	267%
k = 3 cpu (ms) [us]	2.17	4.21	10.1	27.0	92.5	849
k = 1 cpu (ms) [us]	2.28	4.79	12.9	47.2	215	1780
% diff. [us]	5.05%	13.6%	28.7%	74.4%	132%	109%

Figure 17: Comparison of kd-tree culling with different numbers of extents. Cpu times are for the average case.

toothpicks. For the unit scale trial the velocity measured at one toothpick end and averaged over all frames and toothpicks was 0.117% S per frame (image space) and 0.269% W per frame (world space). This amounts to an average of 14.2 and 6.2 seconds to traverse the image or cube side respectively at a 60Hz frame rate.[4]

In a third experiment (Figure 17), we compared performance of the algorithm using kd-trees that sort by different numbers of extents. The same simulations were run as in the first experiment, either exactly as before (k = 3, using two angular extents and the perpendicular camera-dependent spatial extent Z), or using kd-tree partitioning only in a single dimension (k = 1, using only Z). In the second case, the two angular extents were still used for occlusion culling, but not for kd-tree partitioning. This roughly simulates the operation of the NNS algorithm, which first examines objects that overlap in depth before applying further culls using screen bounding boxes. It can be seen that simultaneously searching all dimensions is much preferable, especially as the number of objects increases. For example, in the uniform density case, using a single direction rather than three degrades performance by 14% for 100 objects, 28% for 200, 65% for 400, up to 267% for 1600. The differences in the uniform scale case are still significant but less dramatic, since occlusion culling forms a less important role than layer reordering and occlusion cycle detection.

We used the visibility sorting results to create a depth of field blurred result using compositing operations as described in Section 7, and compared it to a version created with 21 accumulation buffer passes. The results are shown in Figure 18. For the visibility compositing result, a constant blur factor was determined from the circle of confusion at the centroid of the object or object group, for all objects except the cube sides. Because of the large depth extent of the cube sides, these few parts were generated using the accumulation buffer technique on the individual layer parts and composited into the result with the rest.

Canyon Flyby

The second results involve a set of aircraft flying in formation inside a winding valley (Figure 19). We pre-split the valley terrain (see Section 6) into split parts using 2D grids of separating planes and an inflation factor of 20%. The animation involves six aircraft each divided into six parts (body, wing, rudder, engine, hinge, and tail); polygon counts are given in the table below:

object	polygons	hull polygons
body	1558	192
engine	1255	230
wing	1421	80
tail	22	22
rudder	48	28
hinge	64	46
sky (sphere)	480	-
terrain (unsplit)	2473	-

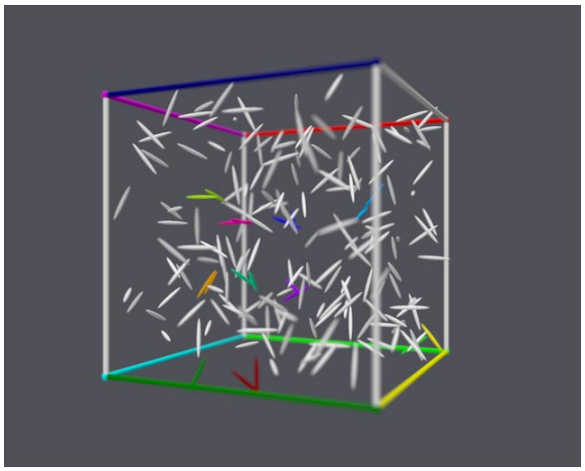
Using terrain splitting grids of various resolutions, we investigated rendering acceleration using image-based interpolation of part images. The following table shows average polygon counts per split part for terrain splits using 2D grids of 20×20, 14×14, 10×10, 7×7, and 5×5:

grid	split objects	polygons/object	hull polygons/object
20 × 20	390	31.98	29.01
14 × 14	191	48.91	37.78
10 × 10	100	76.48	42.42
7 × 7	49	130.45	57.51
5 × 5	25	225.32	72.72

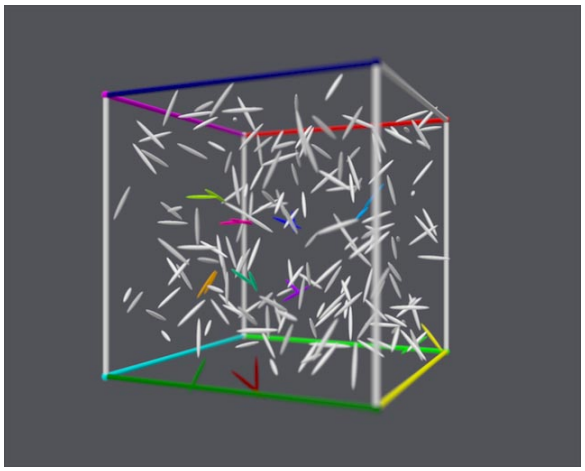
Note that the "polygons" and "polygons/object" column in the above tables are a measure of the average rendering cost of each part, while the "hull polygons" and "hull polygons/object" column is an indirect measure of computational cost for the visibility sorting algorithm, since it deals with hulls rather than actual geometry.

Following results from [Lengyel97], we assumed the 6 parts of each aircraft required a 20% update rate (*i.e.*, could be rendered every fifth frame and interpolated the rest), the terrain a 70% update rate, and the sky a 40% update rate. These choices produce a result in which the interpolation artifacts are almost imperceptible. To account for the loss

[4] While this baseline may seem somewhat slow-moving, it should be noted that small movements of the parts in this simulation can cause large changes in their occlusion graph with attendant computational cost. We believe this situation to be more difficult than typical computer graphics animations. Stated another way, most computer graphics animations will produce similar occlusion topology changes only at much higher velocities.



(a) Accumulation buffer (21 passes)



(b) Visibility compositing

Figure 18: Comparison of depth of field generation methods: The images show two different depth of field renderings from the tumbling toothpicks experiment. Toothpicks comprising a multi-object layer share a common color; singleton layers are drawn in white. Note the occlusion relationships between the sphere/cylinder joints at the cube sides, computed using the algorithm in Appendix B. While pairs of spheres and cylinders are sometimes mutually occluding, the algorithm is able to prevent any further occlusion cycle growth.

Figure 19: Scene from canyon flyby: computed using image compositing of sorted layers with 14 $\times$ 14 terrain split. The highlighted terrain portion is one involved in a detected occlusion cycle with the ship above it, with respect to the bounding convex hulls.

of coherence which occurs when parts are aggregated into a single layer, we conservatively assumed that all parts so aggregated must be rendered every frame (100% update rate), which we call the *aggregation penalty*. The results are summarized in Figure 20.

The column "cpu" shows average and maximum cpu time per frame in milliseconds. The next column ("terrain expansion factor") is the factor increase in number of polygons due to splitting and overlap; this is equal to the total number of polygons in the split terrain divided by the original number, 2473. The next columns show the fraction of visible layers that include more than one part ("aggregate layers fraction"), followed by the fraction of visible parts that are aggregated ("aggregated parts fraction"). Visible in this context means not outside the viewable volume. Average re-rendering (update) rates under various weightings and assumptions follow: unit weighting per part with and without the aggregation penalty ("update rate, unit weighting, (agg)" and ". . . (no agg)"), followed by the analogs for polygon number weighting. Smaller rates are better in that they indicate greater reuse of image layers through interpolation and less actual rendering. The factors without aggregation are included to show how much the rendering rate is affected by the presence of undecomposable multi-object layers. The polygon-weighted rates account for the fact that the terrain has been decomposed into an increased number of polygons. This is done by scaling the rates of all terrain objects by the terrain expansion factor.

In summary, the best polygon-weighted reuse rate in this experiment, 38%, is achieved by the 14 $\times$ 14 split. Finer splitting incurs a penalty for increasing the number of polygons in the terrain, without enough payoff in terms of reducing aggregation. Coarser splitting decreases the splitting penalty but also increases the number of layer aggregations, in turn reducing the reuse rate via the aggregation penalty. Note the dramatic increase from 7 $\times$ 7 to 5 $\times$ 5 in poly-weighted update rate with aggregation penalty (second rightmost column) – splits below this level fill up concavities in the valley too much, greatly increasing the portion of aggregated objects.

It should be noted that the reuse numbers in this experiment become higher if the fraction of polygons in objects with more coherence (in this case, the aircraft) are increased or more such objects are added. Allowing independent update of the terrain's layers would also improve reuse, although as pointed out in [Lengyel97] this results in artificial apparent motion between terrain parts.

9 Conclusion

Many applications exist for an algorithm that performs visibility sorting without splitting, including rendering acceleration, fast special effects generation, animation design, and incorporation of external image streams into a synthetic animation. These techniques all derive from the observation that 2D image processing is cheaper than 3D rendering and often suffices. By avoiding unnecessary splitting, these techniques better exploit the temporal coherence present in most animations, and allow sorting at the level of objects rather than polygons. We have shown that the non-splitting visibility sorting required in these applications can be computed in real-time on PCs, for scenes of high geometric and occlusion complexity, and demonstrated a few of the many applications.

Much future work remains. Using more adaptive ways of splitting container objects is a straightforward extension. Incorporation of space-time volumes would allow visibility-correct motion blur using 2D image processing techniques. Further work is needed to incorporate visibility sorting in animation design systems allowing preview of modifications in their complete context without re-rendering unmodified elements. Opportunities also exist to invent faster and less conservative occlusion tests for special geometric cases. Finally, further development is needed for fast hardware which exploits software visibility sorting and performs 3D rendering and 2D real-time image operations, such as compositing with multiple image registers, blurring, warping, and interpolation.

Acknowledgments

We thank the Siggraph reviewers for their careful reading and many helpful suggestions. Jim Blinn suggested the "sum of outs" resolution for nonbinary cycles. Brian Guenter provided a most helpful critical reading. Susan Temple has been an early adopter of a system based on these ideas and has contributed many helpful suggestions. Jim Kajiya and Conal Elliot were involved in many discussions during the formative phase of this work.

split	cpu (ms)		terrain	layers agg.	parts agg.	update rate			
						unit weighting		poly weighting	
	avg	max	expan. fac.	fraction	fraction	(agg)	(no agg)	(agg)	(no agg)
20×20	17.33	29.82	5.04	0.1%	0.2%	58.3%	58.1%	41.8%	40.7%
14×14	8.08	14.59	3.78	0.4%	1.0%	51.3%	50.2%	38.0%	36.0%
10×10	5.17	9.88	3.09	1.9%	6.4%	48.7%	40.9%	40.4%	30.7%
7×7	4.51	9.68	2.58	5.2%	22.5%	51.9%	37.8%	42.4%	27.8%
5×5	5.37	11.01	2.28	13.0%	53.1%	71.5%	32.3%	72.0%	26.1%

Figure 20: Canyon flyby results.

References

[Appel67] Appel A., "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," In *Proceedings of the ACM National Conference*, pp. 387-393, 1967.

[Baraff90] Baraff, David, "Curved Surfaces and Coherence for Non-Penetrating Rigid Body Simulation," *Siggraph '90*, August 1990, pp. 19-28.

[Bentley75] Bentley, J.L., "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, 18(1975), pp. 509-517.

[Chen96] Chen, Han-Ming, and Wen-Teng Wang, "The Feudal Priority Algorithm on Hidden-Surface Removal," *Siggraph '96*, August 1996, pp. 55-64.

[Chung96a] Chung, Kelvin, and Wenping Wang, "Quick Collision Detection of Polytopes in Virtual Environments," *ACM Symposium on Virtual Reality Software and Technology 1996*, July 1996, pp. 1-4.

[Chung96b] Chung, Tat Leung (Kelvin), "An Efficient Collision Detection Algorithm for Polytopes in Virtual Environments," M. Phil Thesis at the University of Hong Kong, 1996 [www.cs.hku.hk/ tlchung/collision_library.html].

[Cohen95] Cohen, D.J., M.C. Lin, D. Manocha, and M. Ponamgi, "I-Collide: An Interactive and Exact Collision Detection System for Large-Scale Environments," *Proceedings of the Symposium on Interactive 3D Graphics,*, 1995, pp. 189-196.

[Cook84] Cook, Robert, "Distributed Ray Tracing," *Siggraph '84*, July 1984, pp. 137-145.

[Durand97] Durand, Fredo, George Drettakis, and Claude Puech, "The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool," *Siggraph '97*, August 1997, pp. 89-100.

[Fuchs80] Fuchs, H., Z.M. Kedem, and B.F. Naylor, "On Visible Surface Generation by A Priori Tree Structures," *Siggraph '80*, July 1980, pp. 124-133.

[Funkhouser92] Funkhouser, T.A., C.H. Sequin, and S.J. Teller, "Management of Large Amounts of Data in Interactive Building Walkthroughs," *Proceedings of 1992 Symposium on Interactive 3D Graphics*, July 1991, pp. 11-20.

[Gilbert88] Gilbert, Elmer G., Daniel W. Johnson, and S. Sathiya A. Keerthi, "A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space," IEEE Journal of Robotics and Automation, 4(2), April 1988, pp. 193-203.

[Greene93] Greene, N., M. Kass, and G. Miller, "Hierarchical Z-buffer Visibility," *Siggraph '93*, August 1993, pp. 231-238.

[Haeberli90] Haeberli, Paul, and Kurt Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering," *Siggraph '90*, August 1990, pp. 309-318.

[Kay86] Kay, Tim, and J. Kajiya, "Ray Tracing Complex Scenes," *Siggraph '86*, August 1986, pp. 269-278.

[Lengyel97] Lengyel, Jed, and John Snyder, "Rendering with Coherent Layers," *Siggraph '97*, August 1997, pp. 233-242.

[Maciel95] Maciel, Paolo W.C. and Peter Shirley, "Visual Navigation of Large Environments Using Textured Clusters," *Proceedings 1995 Symposium on Interactive 3D Graphics*, April 1995, pp. 95-102.

[Mark97] Mark, William R., Leonard McMillan, and Gary Bishop, "Post-Rendering 3D Warping," *Proceedings 1997 Symposium on Interactive 3D Graphics*, April 1997, pp. 7-16.

[Markosian97] Markosian, Lee, M.A. Kowalski, S.J. Trychin, L.D. Bourdev, D. Goodstein, and J.F. Hughes, "Real-Time Nonphotorealistic Rendering," *Siggraph '97*, August 1997, pp. 415-420.

[Max85] Max, Nelson, and Douglas Lerner, "A Two-and-a-Half-D Motion-Blur Algorithm," *Siggraph '85*, July 1985, pp. 85-93.

[McKenna87] McKenna, M., "Worst-Case Optimal Hidden Surface Removal," *ACM Transactions on Graphics*, 1987, 6, pp. 19-28.

[Ming97] Ming-Chieh Lee, Wei-ge Chen, Chih-lung Bruce Lin, Chunag Gu, Tomislav Markoc, Steven I. Zabinsky, and Richard Szeliski, "A Layered Video Object Coding System Using Sprite and Affine Motion Model," *IEEE Transactions on Circuits and Systems for Video Technology*, 7(1), February 1997, pp. 130-145.

[Molnar92] Molnar, Steve, John Eyles, and John Poulton, "PixelFlow: High-Speed Rendering Using Image Compositing," *Siggraph '92*, August 1992, pp. 231-140.

[Mulmuley89] Mulmuley, K., "An Efficient Algorithm for Hidden Surface Removal," *Siggraph '89*, July 1989, pp. 379-388.

[Naylor92] Naylor, B.F., "Partitioning Tree Image Representation and Generation from 3D Geometric Models," *Proceedings of Graphics Interface '92*, May 1992, pp. 201-212.

[Newell72] Newell, M. E., R. G. Newell, and T. L. Sancha, "A Solution to the Hidden Surface Problem," *Proc. ACM National Conf.*, 1972.

[Ponamgi97] Ponamgi, Madhav K., Dinesh Manocha, and Ming C. Lin, "Incremental Algorithms for Collision Detection between Polygonal Models," IEEE Transactions on Visualization and Computer Graphics, 3(1), March 1997, pp 51-64.

[Porter84] Porter, Thomas, and Tom Duff, "Compositing Digital Images," Siggraph '84, July 1984, pp. 253-258.

[Potmesil81] Potmesil, Michael, and Indranil Chakravarty, "A Lens and Aperture Camera Model for Synthetic Image Generation," *Siggraph '81*, August 1981, pp. 389-399.

[Potmesil83] Potmesil, Michael, and Indranil Chakravarty, "Modeling Motion Blur in Computer-Generated Images," *Siggraph '83*, July 1983, pp. 389-399.

[Regan94] Regan, Matthew, and Ronald Pose, "Priority Rendering with a Virtual Address Recalculation Pipeline," *Siggraph '94*, August 1994, pp. 155-162.

[Rokita93] Rokita, Przemyslaw, "Fast Generation of Depth of Field Effects in Computer Graphics," *Computers and Graphics*, 17(5), 1993, pp. 593-595.

[Schaufler96] Schaufler, Gernot, and Wolfgang Stürzlinger, "A Three Dimensional Image Cache for Virtual Reality," *Proceedings of Eurographics '96*, August 1996, pp. 227-235.

[Schaufler97] Schaufler, Gernot, "Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes," in *Proceedings of the 8th Eurographics Workshop on Rendering '97*, St. Etienne, France, June 16-18, 1997, pp. 151-162.

[Schumacker69] Schumacker, R.A., B. 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, Sept. 1969.

[Sedgewick83] Sedgewick, Robert, *Algorithms*, Addison-Wesley, Reading, MA, 1983.

[Shade96] Shade, Jonathan, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder, "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments," *Siggraph '96*, August 1996, pp. 75-82.

[Sillion97] Sillion, François, George Drettakis, and Benoit Bodelet, "Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery," *Proceedings of Eurographics '97*, Sept 1997, pp. 207-218.

[Snyder97] Snyder, John, and Jed Lengyel, "Visibility Sorting and Compositing for Image-Based Rendering," Microsoft Technical Report, MSR-TR-97-11, April 1997.

[Snyder98] Snyder, John, Jed Lengyel, and Jim Blinn, "Resolving Non-Binary Cyclic Occlusions with Image Compositing," Microsoft Technical Report, MSR-TR-98-05, March 1998.

[Sudarsky96] Sudarsky, Oded, and Craig Gotsman, "Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality," Computer Graphics Forum, 15(3), *Proceedings of Eurographics '96*, pp. 249-258.

[Sutherland74] Sutherland, Ivan E., Robert F. Sproull, and Robert A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, 6(1), March 1974, pp. 293-347.

[Teller91] Teller, Seth, and C.H. Sequin, "Visibility Preprocessing for Interactive Walkthroughs," *Siggraph '91*, July 1991, pp. 61-19.

[Teller93] Teller, Seth, and P. Hanrahan, "Global Visibility Algorithms for Illumination Computations," *Siggraph '93*, August 1993, pp. 239-246.

[Torborg96] Torborg, Jay, and James T. Kajiya, "Talisman: Commodity Realtime 3D Graphics for the PC," *Siggraph '96*, August 1996, pp. 353-364.

[Torres90] Torres, E., "Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes," *Proceedings of Eurographics '90*, Sept. 1990, pp. 507-518.

[Wang94] Wang, J.Y.A., and E.H. Adelson, "Representing Moving Images with Layers," *IEEE Trans. Image Processing*, vol. 3, September 1994, pp. 625-638.

[Zhang97] Zhang, Hansong, Dinesh Manocha, Thomas Hudson, and Kenneth Hoff III, "Visibility Culling Using Hierarchical Occlusion Maps," Siggraph '97, August 1997, pp. 77-88.

A Convex Collision/Occlusion Detection

To incrementally detect collisions and occlusions between moving 3D convex polyhedra, we use a modification of Chung's algorithm [Chung96a, Chung96b]. The main idea is to iterate over a potential separating plane direction between the two objects. Given a direction, it is easy to find the extremal vertices with respect to that direction as already discussed in Section 4.1.1. If the current direction D points outward from the first object A, and the respective extremal vertices with respect to D are v_A on object A and v_B on object B,[5] then D is a separating direction if

$$D \cdot v_A < D \cdot v_B.$$

If D fails to separate the objects, then it is updated by reflecting with respect to the line joining the two extremal points. Mathematically,

$$D' \equiv D - 2(R \cdot D)\,R$$

where R is the unit vector in the direction $v_B - v_A$. [Chung96b] proves that if the objects are indeed disjoint, then this algorithm converges to a separating direction for the objects A and B. Coherence is achieved for disjoint objects because the separating direction from the previous invocation often suffices as a witness to their disjointness in the current invocation, or suffices after a few of the above iterations.

While it is well known that collisions between linearly transforming and translating convex polyhedra can be detected with efficient, coherent algorithms, Chung's algorithm has several advantages over previous methods, notably Voronoi feature tracking algorithm ([Ponamgi97]) and Gilbert's algorithm ([Gilbert88]). The inner loop of Chung's algorithm finds the extremal vertex with respect to a current direction, a very fast algorithm for convex polyhedra. Also, the direction can be transformed to the local space of each convex hull once and then used in the vertex gradient descent algorithm. Chung found a substantial speedup factor in experiments comparing his algorithm with its fastest competitors. Furthermore, Chung found that most queries were resolved with only a few iterations (< 4) of the separating direction.

To detect the case of object collision, Chung's algorithm keeps track of the directions from v_A to v_B generated at each iteration and detects when these vectors span greater than a hemispherical set of directions in S^2. This approach works well in the 3D simulation domain where collision responses are generated that tend to keep objects from interpenetrating, making collisions relatively evanescent. In the visibility sorting domain however, there is no guarantee that a collision between the convex hulls of some object pairs will not persist in time. For example, a terrain cell's convex hull may encompasses several objects for many frames. In this case, Chung's algorithm is quite inefficient.

To achieve coherence for colliding objects, we use a variant of Gilbert's algorithm [Gilbert88]. In brief, Gilbert's algorithm iterates over vertices on the Minkowski difference of the two objects, by finding extremal vertices on the two objects with respect to computed directions. A set of up to four vertex pairs are stored, and the closest point to the origin on the convex hull of these points computed at each iteration, using Johnson's algorithm for computing the closest point on a simplex to the origin. If the convex hull contains the origin, then the two objects intersect. Otherwise, the direction to this point becomes the direction to locate extremal vertices for the next iteration. In the case of collision, a tetrahedron on the Minkowski difference serves as a coherent witness to the objects' collision. We also note that the extremal vertex searching employed in Gilbert's algorithm can be made more spatially coherent by caching the vertex from the previous search on each of the two objects and always starting from that vertex in a search query.

[5] Here, v_A maximizes the dot product with respect to D over object A and v_B minimizes the dot product over object B, in a common coordinate system.

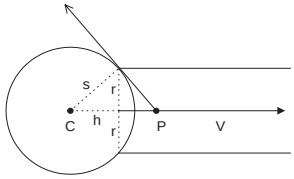


Figure 21: Sphere/cylinder joint occlusion: A cylinder of radius r in direction V is tangent to the sphere of radius s with center C. An occlusion relationship can be derived using the plane through the intersection, at distance h from C along V, and the cone with apex at P such that lines tangent to the sphere through P pass through the circle of intersection.

The final, hybrid algorithm uses up to 4 Chung iterations if in the previous invocation the objects were disjoint. If the algorithm finds a separating plane, it is halted. Otherwise, Gilbert's iteration is used to find a witness to collision or find a separating plane. In the case in which Chung iteration fails, Gilbert's algorithm is initialized with the 4 pairs of vertices found in the Chung iterations. The result is an algorithm which functions incrementally for both colliding and disjoint objects and requires only a single query on geometry that returns the extremal vertex on the object given a direction.

The algorithm can be used to detect collisions between two convex polyhedra, or for point inclusion queries (*i.e.*, single point vs. convex polyhedron). It can also be used for occlusion detection between convex polyhedra given an eye point E. To detect whether $A \rightarrow B$, we can test whether $B' \equiv$ convex_hull($B \bigcup E$) intersects with A. Fortunately, there is no need to actually compute the polytope B'. Instead, the extremal direction search of B' is computed by first searching B as before. We then simply compare that result with the dot product of the direction with E to see if is more extreme and, if so, return E.

B Occlusion Testing for Sphere/Cylinder Joint

This section presents an method for testing occlusion between a sphere and a cylinder tangent to it with respect to its end cap. Let the sphere have center at C and radius s. The cylinder has unit-length central axis in direction V away from the sphere, and radius r, $r \leq s$. Note that the convex hulls of such a configuration intersect (one cylindrical endcap is entirely inside the sphere), and thus the methods of Section 4.2 always indicate mutual occlusion. However, two exact tests can be used to "split" the cycle, indicating a single occlusion arc between the sphere and cylinder. We assume the eye point E is not inside either object.

The cylinder occludes the sphere (and not vice versa) if the eye is on cylinder side of endcap plane; *i.e.*

$$V \cdot (E - C) - h \geq 0$$

where E is the eye point, and where $h \equiv \sqrt{s^2 - r^2}$ is the distance from C to the plane of intersection.

The sphere occludes the cylinder (and not vice versa) if the circle where the sphere and cylinder intersect is invisible. This can be tested using the cone formed by the apex P along the cylinder's central axis for which emanating rays are tangent to the sphere at the circle of intersection. If the eye point is inside this cone, then the circle of intersection is entirely occluded by the sphere. We define $l \equiv \frac{sr}{h} + h$ representing the distance from P to C; P is thus given by $C + lV$. Then the sphere completely occludes the circle of intersection if

$$(E - P) \cdot (C - P) \geq 0$$

and

$$[(E - P) \cdot (C - P)]^2 \geq (l^2 - s^2)(E - P) \cdot (E - P)$$

where the first test indicates whether E is in front of the cone apex, and the second efficiently tests the square of the cosine of the angle, without using square roots. Note that h and l can be computed once as a preprocess, even if C and v vary as the joint moves.

If both these tests fails, then the sphere and cylinder are mutually occluding.