Visibility Culling

Markus Hadwiger
e9425555@stud1.tuwien.ac.at

Andreas Varga
e9426444@stud1.tuwien.ac.at

Abstract

One of the most important problems in computer graphics is the classification of all objects in a scene as either being at least partially visible or totally invisible. The issue of rapidly identifying entirely invisible objects is of importance in any rendering system, but it is especially crucial when interactive frame update rates are desired.

In an ideal rendering system it would be possible to exactly identify those objects that do not contribute to the generated image in any way at zero cost and, furthermore, to draw only the visible parts of those objects.

Since we do not live in an ideal world, however, we try to make the process of determining all objects that are visible from an arbitrary dynamic viewpoint in a scene as efficient as possible. Thus, we want to cull all objects of a scene which are entirely occluded by other objects, or not even contained in the viewing frustum, in large chunks and with minimum cost per culling operation.

This paper first gives an overview of several issues pertaining to virtually all visibility culling algorithms and then examines some of the most important work in detail.

Additionally, we try to assess how some of these algorithms depend on the availability of a hardware z-buffer and how, if at all, they are practically applicable to software-only systems.

1 Introduction

Although the visibility determination problem at the lowest level is nowadays easily solved with hardware z-buffering in most systems, even the fastest hardware available cannot handle entire scenes consisting of, say, a million polygons at interactive rates. So, the naive approach of simply throwing all polygons in a scene at the hardware just won't work. It is not sufficient to employ visibility algorithms that need to check every single primitive in the scene to resolve visibility. Overall scene complexity ought to have no significant impact on the processing time for a single generated image. Rather, this processing time should depend solely on the complexity of that part of the scene being actually visible. Algorithms with this property are called *output sensitive visibility algorithms* or *visibility culling algorithms*.

In this paper the distinction between objects and polygons in a scene is somewhat put aside, as we often assume objects to consist of polygons and consider objects at either the object level or the polygon level, as appropriate. Despite the emphasis on polygonal scenes, since they are simply the most important ones in interactive systems, some of the concepts and algorithms we discuss are also applicable to other object representations – e.g., implicit surfaces, constructive solid geometry (CSG), or objects modeled as collection of spline surface patches – if only objects' bounding volumes are considered.

Nevertheless, the most advanced and also most recent work mostly considers purely polygonal scenes. Of course, all algorithms are trivially applicable if a non-polygonal scene is converted to an explicit polygonal representation in a preprocessing step. Hence, we do not explicitly consider non-polygonal scene or object representations in our discussion.

There are two large steps which need to be taken in order to prune a given scene to a manageable level of complexity:

The first step is to cull everything not intersecting the viewing frustum at all. Since that part of the scene cannot contribute to the generated image in any way it should be eliminated from processing by the graphics pipeline as early as possible. Additionally, those parts of the scene have to be culled in large chunks, because we do not want to clip everything and conclude that nothing at all is visible afterwards. Thus, there has to be some scheme to group polygons and objects together, ideally in a hierarchical fashion.

The second step, which needs to be taken after all polygons outside the viewing frustum have already been culled, is the elimination of unnecessary overdraw. Especially in complex, heavily occluded scenes[1] the number of polygons contained in the viewing frustum will still be too high to be handled by z-buffering alone. Everything still present at this step is potentially visible, but only with respect to the entire viewing frustum. A large number of polygons, depending on the scene, will be drawn into the frame buffer only to be overdrawn by polygons being nearer to the viewpoint later on. And, depending on drawing order, a large number of polygons will be rasterized by the hardware only to be separately rejected at every single pixel as being invisible, because a nearer polygon has already been drawn earlier on and filled the z-buffer accordingly. So, if those polygons could be identified before they are submitted to the hardware, a huge gain in performance would be possible, again, depending on the scene. In heavily occluded scenes the speedup gained can be enormous. Naturally, the efficiency of this identification process is an important economical factor.

To summarize, polygons have to be culled in essentially two steps. First, everything not even partially contained in the viewing frustum is identified and culled. Second, everything that is entirely occluded by other objects or polygons is culled. The remaining polygons can be submitted to the hardware z-buffer to resolve the visibility problem at the pixel level, or any other algorithm capable of determining exact visibility can be used.

Alternatives to z-buffering for resolving exact visibility are mostly important in software-only rendering systems, though.

2 Basics

2.1 Hierarchical Subdivision

An important scheme employed by virtually every visibility culling algorithm is hierarchical partitioning of space. If a partition high up in such a spatial hierarchy can be classified as being wholly invisible, the hierarchy need not be descended any further at that point and therefore anything subdividing that particular partition to a finer level need not be checked at all. If a significant part of a scene can be culled at coarse levels of subdivision, the number of primitives needing to be clipped against the viewing frustum can be greatly reduced.

Hierarchical subdivision is the most important approach to exploiting spatial coherence in a synthetic scene (see section 2.5).

We are now going to look at some of the most common spatial partitioning schemes employing some sort of hierarchy and their corresponding data structures.

[1] These are scenes with a high *depth complexity*. I.e., if all polygons comprising such a scene are rendered into the graphics hardware's frame buffer as seen from any given viewpoint, pixels are drawn many times over, instead of only once. If no visibility culling is employed each pixel is contained in the projection of many polygons and they all have to be rendered in order to, say, have a z buffer determine which single polygon is really visible at that particular pixel. Hence, a major goal of visibility culling is to reduce a given scene's depth complexity as perceived by most levels of the graphics pipeline.

2.1.1 Hierarchical Bounding Boxes

A simple scheme to impose some sort of hierarchy onto a scene is to first construct a bounding box for each object and then successively merge nearby bounding boxes into bigger ones until the entire scene is contained in a single huge bounding box. This yields a tree of ever smaller bounding boxes which can be used to discard many invisible objects at once, provided their encompassing bounding box is not visible.

However, this simple approach is not very structured and systematic and therefore useful heuristics as to which bounding boxes to merge at each level are difficult to develop. The resulting hierarchy also tends to perform well for certain viewpoints and extremely bad for other locations of a synthetic observer.

To summarize, the achievable speedup for rendering complex scenes by using such a scheme alone for visibility culling is highly dependent on the given scene and, worse, on the actual viewpoint, generally rather unpredictable and therefore simple hierarchical bounding boxes are not very useful as sole approach to visibility culling, especially if observer motion is desired. That said, some variant of this idea can nevertheless be very useful when used to provide auxiliary information or setup data; see section 3.3.

2.1.2 Octrees

The octree is a well known spatial data structure that can be used to represent a hierarchical subdivision of three dimensional space. At the highest level, a single cube represents the entire space of interest. A cube for which a certain subdivision criterion is not yet reached is further subdivided into eight equal-sized cubes – their parent cube's octants. This process can be naturally represented by a recursive data structure commonly called octree. Each node of an octree has from one to eight children if it is an internal node; otherwise it is a leaf node.

In the context of visibility culling, octrees are usually used to hierarchically partition object or world space, respectively. For example, each object could be associated with the smallest fully enclosing octree node. Culling against the viewing frustum can then be done by intersecting frustum and octree, culling everything contained in non-intersecting nodes. This intersection operation can also easily exploit the hierarchy inherent in the octree. First, the root node is checked for intersection. If it intersects the viewing frustum, its children are checked and this process is repeated recursively for each node. A node not intersecting the viewing frustum at all can be culled together with its entire suboctree.

This scheme can also be employed to cull entire suboctrees occluded by other objects, provided such an occlusion check for octree cubes is possible and can be done effectively.

One of the most important efficiency problems with using octrees for spatial subdivision of polygonal scenes is caused by the strictly regular subdivision at each level. Since the possible location of octree nodes is rather inflexible, certain problem cases can occur. Imagine, for instance, a subdivision where each polygon is associated with the smallest enclosing octree cube. This process is very dependent on the location of each of the polygons. If a polygon is located about the center of an octree cube it has to be associated with that particular cube, even if its size is very small compared to the size of the cube itself. Even though schemes to alleviate this problem have been proposed, this and similar problems are inherent shortcomings of a regular subdivision, even if it is hierarchical.

The two dimensional version of an octree is called quadtree and can be used to hierarchically subdivide image space, for instance. Incidentally, image space pyramids and image space quadtrees, although very similar, are not the same. A pyramid always subdivides to the same resolution in all paths, i.e. it cannot adapt to different input data as easily; in particular, it is not possible to use different resolutions for different areas of an image.

For an in-depth explanation of octrees and quadtrees, their properties and corresponding algorithms, as well as detailed descriptions of other hierarchical data structures, see [Sam90a] and [Sam90b].

2.1.3 K-D Trees

K-D trees can generally be used to hierarchically subdivide n-dimensional space. A k-D tree is a binary tree, partitioning space into two halfspaces at each level. Subdivision is always done axial, but it is not necessary to subdivide into two equal-sized partitions. So, – in contrast to octrees, where subdivision at each level is always regular – the selection of a separating hyperplane can depend more on the actual data. Usually one halfspace contains the same number of objects as the other halfspace, or exactly one more. This also ensures that the corresponding binary tree is balanced. Additionally, subdivision is not only axial, but also progresses in a fixed order. First, space is subdivided along the first dimension, then the second, and so on, until all k dimensions have been used up. Then the subdivision proceeds using the first dimension once again, as long as a certain termination criterion for further partitioning has not yet been reached.

K-D trees have a number of applications – e.g., n-dimensional range queries; see [Ber97], for instance –, but the most important areas of application as pertaining to visibility culling are three dimensional k-D trees to hierarchically group objects in object and world space, respectively, and two dimensional k-D trees for image space subdivision.

2.1.4 BSP Trees

Binary Space Partitioning or BSP trees, as detailed in [Fuc80], can be seen as a generalization of k-D trees. Space is subdivided along arbitrarily oriented hyperplanes (planes in 3-D, lines in 2-D, for instance) as long as a certain termination criterion is not yet reached. The recursive subdivision of space into two halfspaces at each step produces a binary tree where each node corresponds to the partitioning hyperplane. Normally, a scene consisting of polygons is subdivided along the planes of these polygons until every polygon is contained in its own node and the children of leaf nodes are empty halfspaces. Every time space is subdivided polygons straddling the separating plane need to be split in two. The newly created halves are assigned to the positive and negative halfspaces, respectively. This leads to the problem that the construction of a BSP tree may introduce $O(n^2)$ new polygons into a scene, although this normally only occurs for extremely unfortunate subdivisions.

Primarily, a BSP tree is not used to merely partition space in a hierarchical fashion; it's a very versatile datastructure that can be used in a number of ways. The most important thereof is exact visibility determination for arbitrary viewpoints. For entirely static polygonal scenes a BSP tree can be precalculated once and its traversal at run time with respect to an arbitrary viewpoint yields a correct back-to-front or front-to-back drawing order in linear time.

At each node it must be determined if the viewpoint lies in the node's positive or negative halfspace, respectively. If the viewpoint is contained in the node's positive halfspace everything in the negative halfspace has to be drawn first. Then, polygons[2] attached to the node itself may be drawn. Last, everything in the positive halfspace is drawn. This process is repeated recursively for each node and yields a correct drawing order for all polygons contained in the tree.

[2] A BSP tree node may contain one or more polygons as long as they are all contained in the node's plane. It's often useful to maintain two lists of polygons for each node. One list contains polygons facing in the same direction as the separating plane, whereas polygons contained in the other list are all backfacing with respect to the separating plane. With such a scheme, backface culling can be applied to entire lists of polygons at the same time.

An important observation to make is that each node implicitly defines a convex polytope, namely the intersection of all the halfspaces induced by those hyperplanes encountered when traversing the BSP tree down to the node of interest. This property is very useful with respect to the subdivision of space into convex *cells*, an operation often needed by visibility culling algorithms. If you have, for instance, a polygonal architectural model, a BSP tree can be used to subdivide the entire model into a set of convex polytopes – the cells – which, in the 3-D case, are convex polyhedrons. For such an application it might be sensible to not attach single polygons to the BSP tree's nodes, but to only attach separating planes to internal nodes and entire convex cells together with their constituting polygons to leaf nodes. The model's polygons would then be contained in the boundaries of those convex cells (their 'walls'). Such a modified BSP tree approach yields explicitly described convex polytopes instead of their implicit counterparts in an ordinary BSP tree.

During the walkthrough phase, a BSP tree may be used to easily locate the convex cell containing the synthetic observer, cull entire subtrees against the viewing frustum, and obtain a correct drawing order for whole cells. The walls of those cells themselves may be drawn in any order, since they correspond to the boundaries of convex polyhedrons and thus cannot obscure one another.

For purposes of visibility culling it can be extremely useful to attach probably axial bounding boxes to each of the BSP tree's nodes, encompassing all children of both subtrees and the node itself. During tree traversal each node's bounding box is checked against the viewing frustum and – provided that no intersection is detected – the entire subtree can be culled.

Also, BSP trees can be effectively combined with the notion of potentially visible sets, see below, to obtain a correct drawing order for the set of potentially visible cells.

2.2 Potentially Visible Sets

Many visibility culling algorithms use the notion of potentially visible sets (PVS), first mentioned by [Air90]. This term denotes the set of polygons or cells in a scene which is potentially visible to a synthetic observer. A polygon or cell contained in the PVS is only potentially visible, which is to say it is not entirely sure that it really can be seen from the exact viewpoint at any specific instant. The reason for this is that most algorithms trade the exact determination of visibility for improvements in computation time and an overall reduction of complexity of the algorithm. The result of a visibility query is not the exact set of visible polygons or cells, but a reasonably tight conservative[3] estimate. For instance, in approaches that precalculate potentially visible sets for each cell of a scene subdivided into convex cells, the viewing frustum cannot be taken into account, since the orientation of the observer at run time is not known beforehand. But, the set of all polygons that can be seen from any viewpoint with any orientation in the convex cell is a conservative estimate for the actual set of visible polygons once the exact position and orientation of the observer are known. Moreover, due to algorithm complexity and processing time considerations, maybe not even the former set is calculated exactly but rather estimated in a conservative manner.

If, as mentioned before, cell-to-cell visibility information is precomputed and attached to each cell, it can be retrieved at run time very quickly to establish a first conservatively estimated PVS. This PVS can subsequently be further pruned by using the then known exact location and orientation of the synthetic observer in a specific cell, say, by simply culling the PVS against the actual viewing frustum.

[3] A conservative estimate is an overestimation of the set of actually visible polygons. Although this estimate may be unnecessarily large, it is guaranteed to be a superset of the set of all visible polygons, i.e. no visible polygons can be misclassified as being invisible.

2.3 Portals

Intuitively, a portal is a non-opaque part of a cell's boundary, e.g. a section of a wall where one can see through to the neighboring cell. If the entire scene is subdivided into cells, the only possibility for a sightline to exist between two arbitrary cells is the existence of a portal sequence that can be stabbed with a single line segment. So, if a synthetic observer in cell A is able to see anything in cell B there must exist a sequence of portals P_i where a single line pierces all of these portals to connect a point in A with a point in B.

Rephrased, if we want to determine if anything located in a given cell (including the cell's walls) is visible, through any portals, from the cell the current viewpoint is contained in, we need to calculate if such a sightline exists.

Another way to tackle the concept of portals is to imagine a portal as an area light source where one wants to determine if any of the portal's emitted light is able to reach any point in a given cell. If this is the case, a sightline between those two portals exists and therefore an observer in cell A is possibly able to see some part or all of cell B.

One use of portals is the offline determination of the PVS for a given cell, namely the set of all cells for which a stabbing sequence from a portal of the source cell to a portal of the destination cell exists, regardless of the exact location of the viewpoint within the source cell. This information can be precomputed for each cell in a model and used at run time to instantly retrieve a PVS for any cell of interest.

Alternatively, the concept of portals can be used to dynamically determine a PVS for any given cell entirely at run time, without the use of any precomputed information.

The next section conceptually compares both of these approaches.

2.4 Static vs. Dynamic Visibility Queries

A very important property of an algorithm dealing with the determination of potentially visible sets is its ability or inability to handle dynamic changes in a scene efficiently. This depends on whether the algorithm computes the PVS offline or dynamically at run time.

If dynamic scene changes need to be accommodated quickly – i.e., without a lengthy preprocessing step – the algorithm has to support dynamic visibility queries.

Naturally, dynamic visibility queries suffer from a performance penalty as opposed to static visibility queries.

First, PVS information cannot simply be retrieved from a precomputed table but has to be computed on the fly. To alleviate this problem some algorithms try to exploit temporal coherence (see next section).

Second, due to processing time constraints, the PVS generated by dynamic visibility queries may be not as tight as the PVS generated by table lookups and pruned to the viewing frustum at run time. This incurs a performance penalty at the rasterization level, as the low level visibility determination – in most cases a z-buffer – has to deal with additional invisible polygons.

One promising approach employed by more recent algorithms is to combine both static and dynamic information to determine a PVS. Some amount of a priori information is precomputed and used to speed dynamic visibility queries and enhance their effectiveness, respectively.

2.5 Spatial and Temporal Coherence

In the attempt to make visibility culling algorithms more effective, it is important to take advantage of any potential coherence as much as possible.

The term spatial coherence subsumes both object space and image space coherence. *Object space coherence* describes the fact that nearby objects and polygons in object space very often belong to the same 'class', say, with respect to their visibility. A hierarchical subdivision of object space can directly take advantage of this coherence. The hierarchy is used to ensure that nearby objects are considered together first – to be only considered at a finer level if their invisibility cannot be proved at the coarser level.

Image space coherence is the analog to object space coherence in the two dimensional, already projected, image of a scene. It can be exploited by a subdivision of image space using, e.g., an image space BSP tree. Alternatively, coherence at the pixel level might also be used to advantage by, say, using the fact that adjacent z-buffer entries normally do not have greatly differing depth values.

Temporal coherence can, for example, be exploited by caching some of the information calculated by an algorithm from one frame to subsequent frames, and using this cached information to make an educated guess as to what parts of the scene will actually be visible in the next frame. Such information can be used directly, or to generate some sort of setup information to enhance the performance of a particular algorithm.

3 Algorithms

3.1 Image Space BSP Trees

This section does not describe any single algorithm, but is mostly based on the work presented in [Nay92]. A very interesting application of BSP trees with respect to visibility culling is the use of a two dimensional BSP tree for representation of the projected image of a given scene. We will refer to such trees as image space BSP trees.

The basic idea is based on the simple observation that an image which is the projection of polygons in three-space onto a plane is the union of disjoint planar polygons. As such, it can be represented using a 2-D BSP tree, which in this case is best viewed as a hierarchical description of a set of disjoint convex polygons. So, if we are able to generate the image BSP tree describing the image of a given scene as seen from a specific viewpoint, we only need to scan-convert, i.e. sample, this tree to get the desired image (in terms of pixels).

First of all, this enables us to render the entire scene without any overdraw. We have already eliminated any overlapping parts of polygons during the construction of the BSP tree, operating entirely in continuous space, i.e. no discretization is involved prior to the final rasterization. Incidentally, as long as we operate in continuous space, transformations can be applied to our image without loss of precision, aliasing and the like, as emphasized in [Nay92]. The complete elimination of potential overdraw is very important for speeding up the rasterizing, i.e. pixel-oriented, part of the graphics pipeline.

If we are using 2-D BSP trees to represent our projected image, it is only sensible to also represent the entire scene using a 3-D BSP tree. As it turns out, this is also very convenient for the construction of the image tree:

Remember that a BSP tree can be seen as describing convex regions (convex polytopes) that can be flagged as either being entirely solid or empty. Actually, this is only true for leaf nodes. If we regard an internal node, the node's positive and negative halfspaces may be

anything in between. But those halfspaces will be subdivided until we can reach a definitive decision for each region.

We start with a tree representation of the polygon constituting our viewport, i.e. a rectangle, as image space BSP tree. Everything outside the viewport is classified as being entirely solid, whereas the viewport itself is empty. We then traverse the scene BSP tree in front-to-back order, merging each encountered polygon into the current image tree. This can be visualized as merging two BSP trees – the image tree, and a BSP tree representing the current polygon. Since the correct visibility ordering is already established by the scene tree, we can adopt the following approach: For BSP tree merging we perform an ordered union of both trees, meaning that everything already flagged solid in the existing image tree takes precedence over anything in the polygon's tree. This essentially corresponds to clipping the polygon against a *visibility mask* represented by the current image space BSP tree. Everything already covered by the cumulative projection of polygons already processed cannot be occluded by anything encountered during the traversal of the scene BSP tree later on.

This whole process can be considered as being either two dimensional or three dimensional, as appropriate. Essentially, the 2-D image tree can also be visualized as some sort of 3-D *beam-tree*, where each beam (a volume) is constituted by the volume induced by the polygon and the eye point.

But not only can potential overdraw be completely eliminated, the image space BSP tree can be used to effectively cull the scene against the current visibility mask. Culling is easily employed in such an approach, since the merging of the two BSP trees naturally culls entire objects high up in the hierarchy if the entire tree describing an object is completely contained in a solid region of the image space BSP tree.

With respect to rendering hardware there are both advantages and disadvantages to the use of image space BSP trees. First of all, a z-buffer is not required at all, thus it is very conducive to software-only implementations. On the other hand, there is a lot of work involved to resolve the visibility problem and existing hardware is not of much help here.

Furthermore, since the whole scheme is inherently based on the use of a 3-D BSP tree to represent the scene, all restrictions in this regard apply, say, only static scenes can be used efficiently. Perhaps a hybrid approach would be feasible, namely only using the image tree to cull away portions of a scene not represented by a BSP tree itself, by checking objects' bounding volumes against the current visibility mask. This, however, would render the construction of the image BSP tree rather complicated.

[Nay97] is a nice survey of the potential uses of BSP trees for interactive applications, mainly in the entertainment area.

3.2 Hierarchical Z Buffers

The method presented in [Gre93] is a way to gain speedups by exploiting the spatial and temporal coherence of a scene as described in section 2.5. To accomplish this goal, it uses a combination of three different algorithms where each alone provides already a substantial speedup when interactively rendering a large scene, but all three should be combined to gain the full benefit.

In order to take advantage of object space coherence, it builds an object space octree, which is used at rendering time to quickly cull large portions of the scene if an enclosing cube is completely hidden. Apart from the fact that the cube gets intersected with the viewing frustum, it also has the advantage that through its recursive nature, only relevant parts of the scene are checked. If it can be proved that a full cube is hidden, all of its children need not be checked.

The octree is constructed by recursive subdivision of the root cube (which encloses the whole scene) until each cube contains a given maximum of polygons. If a polygon intersects one of the planes that separate the cube into its eight child cubes, the cube gets subdivided.

To render the picture the algorithm checks the visibility of each octree cube recursively and only the geometry that is contained in (at least partially) visible cubes.

The second algorithm used in the Hierarchical Z Buffer Visibility method is the so-called Z pyramid which is used in conjunction with the object space octree. This is a logical extension to the traditional Z buffering algorithm. Instead of using a single Z buffer at the image resolution, it consists of multiple Z buffers, with decreasing resolution. Starting at the full image resolution (the lowest level of the pyramid), four pixels are grouped to form one pixel of the next-coarser pyramid level, thus reducing the resolution to the half. This is continued until the top level of the pyramid, which is a Z buffer with only one pixel, contains the farthest Z of the whole image. To maintain the Z pyramid, Z values have to be propagated from the finest to the coarsest level of the pyramid, until a Z value is found which already is as far away as the new Z value. At that point the propagation can be stopped.

During rendering, the algorithm needs to find the finest pyramid level that contains a pixel covering the image space region enclosing the polygon we want to draw. It then compares the nearest Z value of the polygon with the Z value of that Z buffer pixel. If that Z value is closer than the nearest Z value of the polygon, the polygon is completely hidden. It is not necessary to recurse to finer levels of the Z pyramid, unless the polygon is not fully hidden. This algorithm is used in combination with the object space octree to quickly determine the visibility of the faces of the octree cubes, in addition to the visibility determination of polygons inside the octree cubes.

To exploit temporal coherence in animations or interactive walkthroughs, a third algorithm is presented. It uses a temporal coherence list, which basically contains the visible octree cubes from the previous frame. When a new frame has to be rendered, the cubes on the list are rendered first and marked as rendered, before continuing with the normal algorithms described above. The resulting Z buffer is taken to form the initial Z pyramid. Since there is supposed to be a lot of temporal coherence in walkthrough scenes, most of the visible portions should already be rendered after this step. Therefore the Z pyramid already contains enough information to quickly determine the visibility of the rest of the scene.

After the new frame has been rendered, the Z pyramid is used to eliminate old cubes from the temporal coherence list that are now invisible.

The advantages of the Hierarchical Z Buffer Visibility method include its efficient way of exploiting spatial and temporal coherence, but it’s not possible to fully implement it with current graphics hardware, therefore only hybrid software-hardware approaches can be used.

This method is expanded upon in [Gre94], which uses an image space quadtree instead of the Z pyramid and is able to additionally perform antialiasing within guaranteed error bounds.

3.3 Hierarchical Occlusion Maps

Hierarchical occlusion maps (HOM), as presented in [Zha97], are an alternative way to combine a two dimensional image space hierarchy with a hierarchy of bounding volumes in object space. The work is similar to the approach of [Gre93] in many respects, although different trade-offs were made.

First of all, the employed occlusion maps do not contain any depth information. Instead, a pyramid of opacity values corresponding to a set of so called occluders is built. An *occluder* is a large polygon located near the viewpoint and therefore having high probability of occluding a large part of the scene. These occluders are retrieved from a database built offline, i.e. as a

preprocessing step, called the occluder database. At run time, a set of potential occluders with respect to the current viewpoint is extracted from this database. Culling against the viewing frustum is done using a simple bounding volume hierarchy in object space. Then, all remaining occluders are rendered into an occlusion buffer, where each pixel belonging to the projection of an object is considered opaque and all other pixels are considered transparent. The occlusion buffer is then filtered to successively coarser levels – the buffer itself becoming the finest level of a pyramid of *occlusion maps*. Filtering in this case means simply averaging adjacent pixels to get opacity values – the ratio of opaque area to the entire area covered by the averaged square. Potential *occludees* (everything not being considered as occluder) are checked against these occlusion maps starting at the coarsest level of the HOM pyramid. If an occludee bounding volume's screen space projection is covered by an entirely opaque area it can be culled. If not, the algorithm continues at the next finer level.

A notable property of the HOM algorithm is that it makes approximate visibility culling possible. Since the entries in an occlusion map represent the opacity of everything encompassed in finer levels of resolution, alpha thresholds can be used to regard everything above a given degree of opacity as completely opaque with respect to occlusion.

Also, the HOM approach is more conducive to implementation using current graphics hardware than are hierarchical z-buffers. Standard texture mapping hardware capable of performing bilinear filtering operations can be used to speed up the construction of the occlusion map pyramid. Hence, this method's equivalent to the very expensive step of performing z-value propagation through an entire hierarchy – as needed for the z-pyramid used by [Gre93] – may be not nearly as expensive.

3.4 Precalculated Potentially Visible Sets

The PVS approach described in section 2.2 uses a precalculation step to determine a conservative estimate for the visibility problem in interactive walkthrough scenes. However, the exact way of determining the cell-to-cell visibility is part of the implementation. A possible method for precalculating the potentially visible sets for a given scene is presented in [Tel91], although the algorithm described there is limited to cells which consist only of faces parallel to the axis-planes. An improved method can be found in [Tel92].

The precomputation algorithm divides the scene into cells, using a BSP tree, whose splitting planes contain the major axial faces. Since these faces are all axis-parallel, the BSP tree can be considered as a k-D tree with k=2. The root cell of the k-D tree corresponds to the bounding box of the full scene. Now each face F of the cell is classified as either:

- disjoint, if F has no intersection with the cell
- spanning, if F partitions the cell into components that intersect only on their boundaries
- covering, if F lies on the cell boundary and intersects the boundary's relative interior
- incident, if F lies in the cell, and does not intersect with any spanning or covering faces

See Figure 1.

Using this classification and a set of heuristics based on it, the scene is subdivided into cells. Each cell can contain one or more non-opaque faces, called *portals,* which are stored with the cell, and are used to construct an adjacency graph of cells. Each vertex in this graph represents a cell, and an edge between two vertices means that there is a portal between those two cells. This graph is used to find direct neighboring cells, which have trivial cell-to-cell visibility.

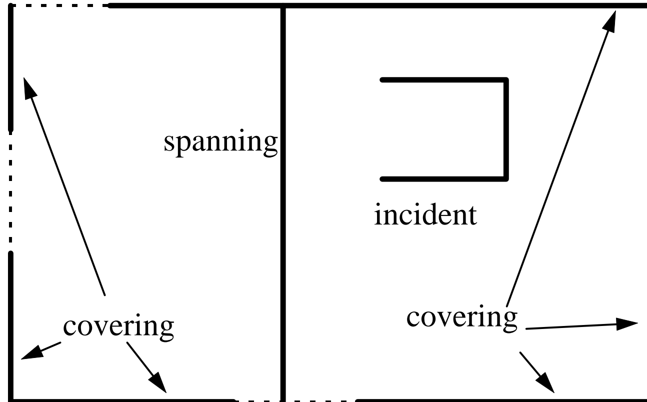

Fig. 1: Input face classifications

For the non-trivial case, the algorithm has to find a sightline through a sequence of portals, to determine the visibility between two cells that are not direct neighbors. To do this it tries to find portal sequences that can be stabbed by sightlines, such as in Figure 2.

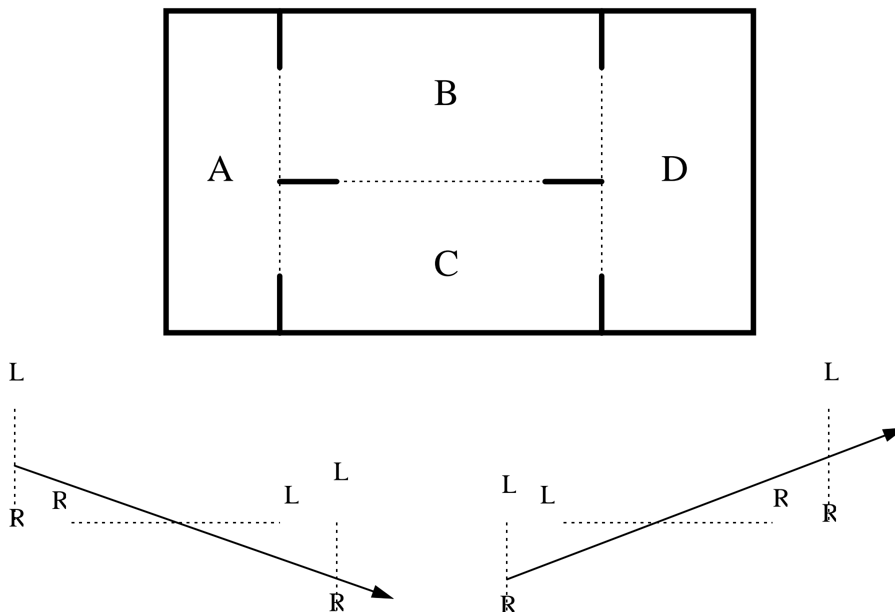

Fig. 2: Two oriented portal sequences admitting two sightlines between cells A and D

Portal sequences that admit sightlines are generated by a recursive depth-first search in the adjacency graph, and stored in a so-called stab tree.

The endpoints of each portal in a portal sequence are separated in the set of left endpoints (L) and right endpoints (R), and to determine if a sightline can stab this portal sequence, it has to be proved that the sets L and R are linearly separable. This is a linear programming problem that can be solved in linear time, i.e. O(n) for n faces in a scene.

The final result of the algorithm, the cell-to-cell visibility is then calculated by recursively testing all portal sequences for a stabbing sightline originating at a given cell C. This will produce a set of cells that are potentially visible from any point in cell C.

A comprehensive analysis of an application of this method is found in [Fun96].

3.5 Dynamic Visibility Culling with Portals and Mirrors

If dynamic visibility processing is required, a lengthy precalculation algorithm such as in section 3.3 cannot be used. Instead a more conservative but faster method has to be developed. [Lueb95] presents such a fast method for dynamically querying the visibility of a scene, with the position and orientation of the observer given. Similar to the static approach described above, the scene is subdivided into a number of cells, with portals connecting those cells. To determine the visibility, the algorithm projects the vertices of each portal into screen space and calculates the axial 2D bounding box of the projected portal. This 2D box is called the *cull box*. An object whose screen space projection lies outside of the cull box, is not visible through the portal, and therefore does not need to be rendered.

A recursive method is used to traverse from each cell to the next. At each step, the cull box of the current portal is compared with the intersection of all cull boxes of all the previous portals that have been visited, to find out if there is a potential sightline through that sequence of portals (see Figure 3). If the cull box of the current portal lies outside of the aggregated cull boxes, we can be sure that there is no possibility to see further through this portal, and therefore we can traverse back to the previous portal, continuing the recursive search. During that traversal, objects in the cells are also checked against the intersection of cull boxes, and are only rendered if they are lying at least partially inside. An object that gets rendered is marked, so that it won't be rendered at a later point of the recursive traversal yet again.

Another possibility is the clipping of each object against the aggregate cull box of the portal sequence.

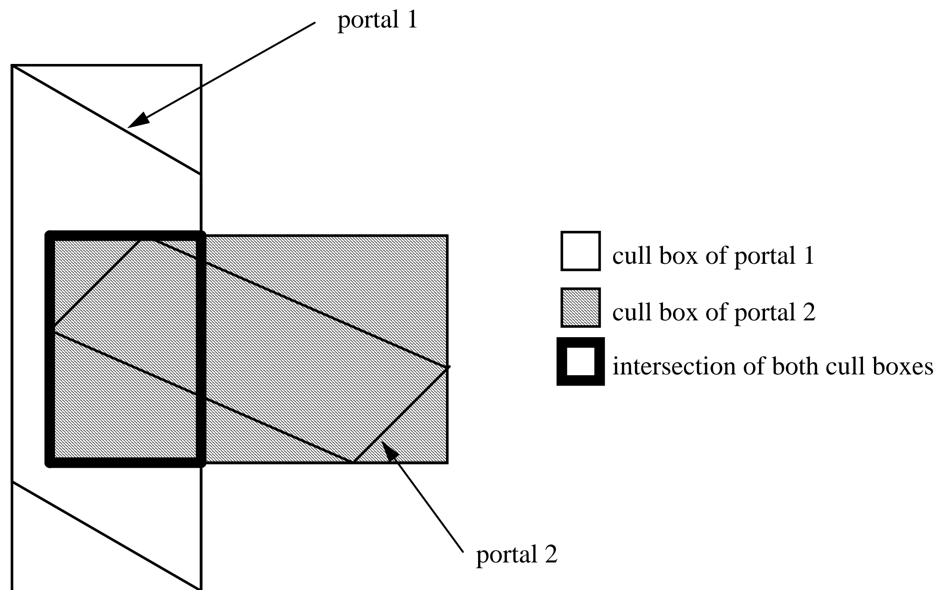

Fig. 3: Intersecting the cull boxes of two portals

Since mirrors can be considered as portals into a reflected world, they can be treated like ordinary portals, in terms of visibility calculation. This, however, brings up some difficulties and generally does not work well with scenes that are dynamically changed.

3.6 Occlusion Culling By Identifying Large Occluders

An inherent restriction of many visibility culling techniques needing to subdivide an entire scene into a collection of cells, i.e. polyhedrons, is their inapplicability to scenes with huge empty spaces, e.g. outdoor scenes. In such scenes, the identification of portals through which a

synthetic observer is able to see is not really feasible, since most of the area would be covered by such portals. These algorithms depend tremendously on heavy occlusion throughout an entire scene.

The work in [Coo97] uses the identification of large occluders near the current viewpoint, a similar idea as used in the HOM approach of section 3.3. The determination if an object is actually occluded by such an occluder is done in a more analytical, continuous space approach, though. Using the viewer's location, the occluder, and the potential occludee, the notion of *supporting* and *separating planes* is introduced. Using these planes one can analytically determine if the occludee is actually not reachable via a sightline from the viewpoint.

This approach depends heavily on the efficiency of the selection of occluders for each frame. Furthermore, potential occlusion of course cannot be calculated for every possible occluder-occludee pair, so the scene is subdivided using a k-D tree to exploit spatial coherence. To alleviate the potential bottleneck of always recomputing the needed supporting and separating planes, temporal coherence is used by caching this information for each k-D tree node encountered during traversal.

The notion of computing only a conservative estimate for the set of actually visible primitives prevalent in this algorithm demands exact visibility computation be performed by lower levels of the graphics pipeline, e.g. a hardware z-buffer.

3.7 Other Relevant Work

[Sud96] contains a very nice survey of visibility culling techniques and goes into considerable detail on how dynamic objects can be accommodated in some of the already known approaches. The paper describes how an octree can be updated effectively when objects in its domain are moving around and briefly considers dynamic updates of BSP trees.

Most importantly, it introduces the notion of *temporal bounding volumes* (TBVs). If the motion and animation of an object are known beforehand, i.e. somewhat constrained, temporal bounding volumes are easy to calculate. The TBV simply encompasses the entire area where an object is allowed to be located at any given time. If this is not possible, the problem can be tackled differently. For instance, if at least some information, say, maximum velocity is known for an object, TBVs can be calculated for certain periods of time. During such a period the bounding volume must not be violated, i.e. the object must be guaranteed to not leave its interior lest visibility errors might occur.

[Yag96] uses a strictly regular subdivision method to decompose highly irregular environments, dubbed caves in the paper. A list of intersecting primitives is calculated for each cell and attached to the cell as a preprocessing step. Cells are also labeled as either being solid, empty, or containing walls, at that time. In the walkthrough phase cell-to-cell visibility is computed be regarding only the cells being part of a so called sight corridor between the source and destination cell. In general, it is hard to see how this approach could yield interactive rates for three dimensional models. Nevertheless, application to 2-D data, as presented in the paper, seems feasible due to many simplifications possible compared to the general 3-D case.

[Fal93] contains an interesting discussion of various aspects relating to interactive animation of simulated terrain covering a very wide area. Visibility culling is done using a quadtree subdivision of the elevated height field representing the area. Interestingly, level of detail (LOD) techniques are employed to traverse the quadtree only to a level corresponding to a given square's distance to the viewpoint.

4 Conclusions

In this paper we tried to give a comprehensive, although short, survey of the general area of visibility culling techniques with a strong emphasis on real-time oriented applications.

We described the most important data structures and basic ideas in detail. Since most output sensitive visibility algorithms share some ideas and approaches it is very important to have a firm grasp on the workings of those spatial data structures and subdivision techniques.

We reviewed some of the most important visibility culling methods and briefly mentioned other relevant work.

The problem of rapidly culling the invisible part of a complex scene is still very demanding. There is no definite approach that works for all conceivable kinds of input data. Instead, trade-offs have to be made and knowledge about the particular kind of data is essential. Furthermore, the availability and type or lack of advanced graphics hardware can influence one's choice of algorithm substantially.

Another interesting aspect of an output sensitive visibility algorithm is if the secondary data structures it uses – which oftentimes can be very expensive to maintain, both in terms of processing time and memory consumption – can also be used for other tasks, therefore amortizing in other areas as well. For example, a representation of a scene as 3-D BSP tree can also be used for effective collision detection in virtual reality applications as can any decomposition of a scene into convex cells sporting a corresponding search structure.

References

[Air90] John M. Airey, John H. Rohlf and Frederick P. Brooks, Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *Proceedings 1990 Symposium on Interactive 3D Graphics* (1990). pp. 41-50.

[Ber97] Mark De Berg, et. al. *Computational Geometry: Algorithms & Applications.* Springer-Verlag, 1997.

[Coo97] Satyan Coorg and Seth Teller. Real-Time Occlusion Culling for Models with Large Occluders. In *Proceedings of 1997 Symposium on Interactive 3D Graphics* (1997). pp. 83-90.

[Fal93] John S. Falby, Michael J. Zyda, David R. Pratt and Randy L. Mackey. NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulation. In *Computers & Graphics,* 17(1). (1993). pp. 65-69.

[Fuc80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by A Priori Tree Structures. In *SIGGRAPH '80 Conference Proceedings* (1980). pp. 124-133.

[Fun96] Thomas Funkhouser, Seth Teller, Carlo Séquin, and Delnaz Khorramabadi. The UC Berkeley System for Interactive Visualization of Large Architectural Models. In *Presence,* 5(1). (1996). pp. 13-44.

[Gre93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-Buffer Visibility. In *SIGGRAPH '93 Conference Proceedings* (1993). pp. 231-238.

[Gre94] Ned Greene and Michael Kass. Error-Bounded Antialiased Rendering of Complex Environments. In *SIGGRAPH '94 Conference Proceedings* (1994). pp. 59-66.

[Lueb95] David Luebke and Chris Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proceedings 1995 Symposium on Interactive 3D Graphics* (1995). pp. 105-106.

[Nay92] Bruce F. Naylor. Partitioning Tree Image Representation and Generation from 3D Geometric Models. In *Proceedings of Graphics Interface '92* (1992). pp. 201-212.

[Nay97] Bruce F. Naylor. Interactive Playing with Large Synthetic Environments. In *Proceedings 1997 Symposium on Interactive 3D Graphics* (1997). pp. 107-108.

[Sam90a] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[Sam90b] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1990.

[Sud96] Oded Sudarsky and Craig Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. In *Proceedings of Eurographics '96 Conference* (1996). pp. 249-258.

[Tel91] Seth J. Teller and Carlo H. Séquin. Visibility Preprocessing For Interactive Walkthroughs. In *SIGGRAPH '91 Conference Proceedings* (1991). pp. 61-69.

[Tel92] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis. University of California at Berkeley, 1992.

[Yag96] Ronie Yagel and William Ray. Visibility Computations for Efficient Walkthrough of Complex Environments. In *Presence,* 5(1). (1996). pp. 45-60.

[Zha97] Hansong Zhang, Dinesh Manocha, Tom Hudson and Kenneth E. Hoff III. Visibility Culling using Hierarchical Occlusion Maps. In *SIGGRAPH '97 Conference Proceedings* (1997). pp. 77-88.