Data Structures and Algorithms for Navigation in Highly Polygon-Populated Scenes

Carlos Saona-Vázquez*, Isabel Navazo and Pere Brunet

Abstract

We present the visibility octree, a new data structure to accelerate 3D navigation through very complex scenes. Our approach employs a conservative visibility technique to compute an approximation to the visibility space partition. This approximation is computed and stored hierarchically at a preprocessing stage. We believe its main contribution to be its ability to provide an effective control over the coarseness of the approximation. A preliminary test with some randomly generated indoor scenes seems to show that the visibility octree will perform well on densely occluded scenes.

1 Introduction

This work is focused on the interactive navigation through polygonal models. There is a wide range of applications nowadays whose requirements surpass even the most expensive high-end graphics workstations. To name a few, ship design, architectural design and virtual reality applications have hundreds of thousands or even millions of polygons. Current graphics hardware is not able to cope with this kind of scenes at interactive frame rates. In recent years, many 3D navigation algorithms have been developed to lessen the limitations of current workstations.

The problem can be specified as follows: given a set P of static polyhedrical objects, we want to compute the set V of visible polygons from every possible viewpoint. Moreover, we want this computation to be done quickly, in constant or at least logarithmic time. As a matter of fact, the perfect algorithm would take as an input the desired frame rate and would be able to guarantee it. No such algorithm is known yet.

In section 2, we will sketch briefly the theoretical nature of the problem and study its complexity. Next, in section 3 we will survey previous related work and classify them according to various criteria. Section 4 describes our contribution, the visibility octree, and shows some preliminary results. Finally, section 5 analyzes them and ends with a plan of future work.

---

* This work has been supported by an FPI grant from the Ministerio de educación y ciencia (Spanish Ministry of Education and Science) and the TIC-95-630 project

2 Theoretical frame

Both the computer graphics and the computational geometry communities have worked in the area of visibility for decades. The first problem they faced was the visibility computation from a fixed viewpoint. It can be stated informally as follows. Given a set of modeling primitives (usually polyhedral or polygonal) and a viewing frustum that defines the observer position and its field of view, compute the set of visible polygons. Note that polygons in this set need not correspond to input polygons.

This problem was intensively addressed in the seventies, and many algorithms were developed. There is a classic paper by Sutherland, Sproull and Schumacker that not only describes all those early works but also introduces a classification scheme [21]. There is a more recent survey in [8]. However, due to the decreasing cost of memory and the development of VLSI technologies, the visibility algorithm that became standard was the simple, memory intensive, non-sophisticated z-buffer algorithm.

A further step is the computation of global visibility. The goal is an algorithm that computes visibility from every possible viewpoint at once. This concept naturally induces a visibility partition of space. Works in this field study the complexity of this partition.

Before proceeding further, we need some theoretical background. For a start, we will define an equivalence relation between images using the image structure graph (see [12]). The edges and vertices in this graph correspond to line segments and endpoints in the image. Besides, vertices are labeled with the names of object edges whose projections meet at them; every graph edge is labeled with the labels of its two adjacent vertices. We will say that to images are topologically equivalent or that they have the same aspect iff their corresponding image structure graphs are isomorphic.

Now we can define another equivalence relation but this time between points of $\mathcal{R}^3$. Two (view) points v and w are equivalent iff there is a continuous curve between them such that the images of each point in the curve (including v and w) are topologically identical. This equivalence relation allows us to split $\mathcal{R}^3$ into regions of constant aspect[1]. This partition is called the Visibility Space Partition, VSP hereafter. A boundary of a VSP regions is called a visual event (VE for short) for it marks a change in visibility. The VSP is the dual of another structure called the aspect graph, which the image recognition community has studied throughly for two decades. The vertices of this graph correspond to regions of the VSP and its edges connect vertices whose regions are adjacent.

Note that the regions of the VSP are not maximal regions but maximal connected regions. Thus, it is possible for the VSP to contain non adjacent regions whose images are topologically identical. Moreover, it is possible that different regions exist such that their sets of visible polygons are equal.

---

[1] Strictly, this statement does not hold because a perspective view is usually defined with more parameters than a single viewpoint (viewing direction, upward direction, etc.). But this is a minor detail that can be fixed up by using spherical projection instead of planar projection.

Anyway, it is clear that the set of visible primitives remains constant when moving within a region of the VSP. Therefore, it is possible to build a data structure that represents a map of $\mathcal{R}^3$ that associates every viewpoint to the set of its visible polygons. This data structure could be a labeled, enriched version of the aspect graph. The label of each edge in the graph would store the boundary that has to be crossed to pass from one region to another; on the other hand, each vertex would keep the set of the visible primitives of its associated VSP region. Thus, the computer could keep track of the user's location and the visible primitives very easily by checking this boundaries.

Unfortunately, this algorithm is not feasible due to the combinatorial complexity of the VSP, in the first place, and to the computational complexity of some of its boundaries, in the second place. It follows a brief description of the boundaries of the VSP.

Visual events can be of three different types: vertex-edge (VE), edge-edge-edge (EEE) and horizon (for a demonstration, see [17]). Vertex-edge events are the locus of points p such that there is a line thru p that intersect a vertex and a edge. It is clear that this boundary is a plane in $\mathcal{R}^3$. The EEE visual event is the locus of all the lines that intersect the three edges. This boundary is a ruled quadric surface. Finally, horizon events correspond to the supporting planes of the polyhedra's faces.

It can be proved (again, see [17]) that this boundaries yield a spatial complexity of $\Theta(n^9)$ for the case of a VSP with non-convex polyhedra and perspective projection, where n is the number of vertices in the scene. Table 1 shows other cases; note that all the bounds are tight. It goes without saying that this size makes infeasible the effective construction of the VSP, specially when the input data has hundreds of thousands of polygons. Besides, the quadric nature of some of its boundaries worsens the problem.

Of course, it could be possible that, even though the bounds are tight, the normal case was much more simpler. But some empirical results seem to show that the usual case is not simpler enough. Plantinga and Dyer proposed in [17] a structure to represent the aspect graph they called the aspect representation (asp). To our knowledge, it has not been implemented for the general case of perspective projection and non-convex polyhedra. The 3D visibility complex of Durand, Drettakis and Puech did not reach the implementation stage either [9]. Only a later work of Durand, Drettakis and Puech, the visibility skeleton, has the virtue of being fully implemented [10]. Though the visibility skeleton is a simplified version of the aspect graph, it is powerful enough to allow for straightforward computation of exact visibility. Its main drawback is that, as reported by the authors, it used more than 400Mb for a visually common scene of roughly 1500 polygons. This figure utterly discards the visibility skeleton if scenes of thousands of polygons are involved. Moreover, it seems to point that the VSP is too complex for scenes of that size.

3 Previous work

We can classify research on this area as pertaining to three main families; namely, multiresolution algorithms, image-based rendering and visibility compu-

	convex polyhedra	non-convex polyhedra
orthographic projection	$\Theta(n^2)$	$\Theta(n^6)$
perspective projection	$\Theta(n^3)$	$\Theta(n^9)$

Table 1: Spatial complexity of the Visual Space Partition

tation algorithms. As usual, each approach has both advantages and drawbacks. Before analyzing them, though, we will introduce some standard terminology and present some classification criteria. We will end the section with table 2 that shows each method's highlights.

3.1 Classification criteria

From now on, we will refer to navigations algorithms as being static or dynamic, image-space or object-space oriented, and as impostor oriented or not. As we will see, neither of these three categories are mutually exclusive.

The first criterion refers to the moment when heavy computation is done. Static algorithms do most of their calculations at a preprocessing stage and before the user actually navigates through the scene. Typically, this computation will last many hours or even several days when state of the art hardware is used. Nevertheless, this time-consuming preprocessing should not become a problem when the scene is a final model that is not going to be altered but going to be navigated many times. A computer-modeled building which is intended to be shown to will-be buyers is a classical example of this kind of application. On the other hand, dynamic algorithms have light preprocessing, if any, and their main computations are performed on the fly, as the user walks thru the scene.

The second criterion reflects whether the work is done on the polyhedral, analytical space, or on the discrete, rastered one. Algorithms in the second group usually suffer from visual artifacts, while those in the former one do from floating-point errors and have to deal with degeneracies in the model.

Last, many methods try to solve the problem by replacing some scene objects with simpler ones, called impostors. These impostors are used instead of the original objects when the algorithm decides that drawing the object in full detail is not worth the effort. Typically, this decision will be made attending to pixel contribution in the final image or to time-bucket restrictions. Moreover, impostors need not be in one-to-one correspondence with original objects. In fact, in many works not every original object has an impostor, and one impostor may replace not just one object but many of them. Finally, impostors do not have to share the polyhedral nature of their original model counterparts; as a matter of fact, many authors employ non-polyhedral, bitmapped impostors.

3.2 Multiresolution algorithms

The standard way to define multiresolution models is to say that they are those models that provide several representations for the objects in the scene but at different resolutions or levels of detail (hence the alternative term LOD models). This multiplicity of representations can be achieved by using simplification techniques on the original objects.

It has become a common practice not to consider image-based rendering as a multiresolution model, although it fits the standard definition. Some authors strengthen the concept by stating that the simplificated models must share the representation scheme with the originals. This rules out image-based rendering because the textured-models are not boundary representations.

Typically, multiresolution algorithms perform a lengthy preprocessing step to compute coarser representations for objects in the scene. At execution time, the computer has to decide about the resolution level it will employ to render each object. Multiresolution algorithms are often classified according to the simplification method they use and to their selection strategy. The reader can find a thorough survey of multiresolution and simplification algorithms in a report from Andújar [3].

The two main disadvantages of this kind of algorithms is connected with their object-based nature. First, they cannot deal with polygons soups. Even if the scene is object-structured, these algorithms do not decrease the total number of objects. This is a bound on the degree of acceleration they can provide.

3.3 Image-based rendering

There are two key concepts to image-based rendering. The first one is the the fact that final, bitmapped images usually differ very little from frame to frame in a typical navigation. The second one is the increasing availability of texture mapping hardware in nowadays graphic workstations. This kind of techniques achieves significant frame-rate improvements by replacing part of the scene's geometry with one or more texture-mapped polygons whose textures correspond somehow to the discarded geometry. This textures need not have been generated from the same viewpoint as the current one. The computer will be able to navigate noticeably faster if the algorithm chooses wisely which parts of the geometry are replaced by textures. Existent algorithms can be classified depending on their answers to the following questions:

- Are impostors associated to objects or to sets of objects?
- Are the bitmaps generated off-line or on-line?
- How does the computer measure the accuracy of a texture?
- How does the computer decide between geometry and images?
- What kind of data structure does the algorithm use to store the scene?

Maciel and Shirley in [15] build one bitmap for each one of the six faces of every node of the object octree they use to store the scene. These textures are computed off-line. Their accuracy is also computed off-line by taking samples at different viewpoints and comparing them with what the texture will look like from those viewpoints. The computer performs this comparison with image-processing techniques and stores the results into a table. The algorithm computes a benefit/rendering-cost ratio to choose between geometry and images. The benefit is a custom function that takes as parameters the size of the objects, their distance to the observer and their semantic relevance in the scene.

Schaufler's algorithm ([19]) is the only one of the works we researched that generates impostors only on a per-object basis. It does not use any special data structure, computes impostors at navigation time and employs an angular distance metric to decide when to recompute texture and whether to use geometry or images. Schaufler's later work with Stürzlinger (see [18]) does use a kd-tree, and generates textures dynamically on a per node basis.

Aliaga's 1996 algorithm in [1] divides the scene uniformly into voxels before navigation takes place. It computes a texture for every voxel that is further away from the current viewpoint than a user-defined threshold distance.

Sillion, Drettakis and Bodelet introduce in [20] a specific algorithm to navigate thru cities. They take profit of the special nature of urban scenes by using a graph to represent the scene. Its nodes are the city streets and its edges are the city blocks. The computer generates off-line two impostors for each street corresponding to the pair of possible vistas along the street. At execution time, the algorithm only uses geometry to render the blocks adjacent to the street the user is located; the rest of the scene is rendered with textures.

Finally, Aliaga and Lastra (see [2]) also present a specific algorithm, this time for interiors of buildings. The scene has to be structured into rooms, doors and windows. At the preprocessing stage, the computer associates several textures to every door and window, each texture generated from a different viewpoint. At running time the algorithm only draws geometry for the objects that are inside the room where the user is.

Advantages

- Most of the works achieve significant speed improvements.
- Except for a few of them, image-based algorithms do not impose any requirements on the scene. They are able to work with polygon soups without any speed diminishment.
- They are able to cope with forest-like scenes very efficiently, unlike visibility algorithms.

Drawbacks

- Very big memory requirements, specially when the images are not computed dynamically.
- The change from geometry-based rendering to texture-based rendering (and vice versa) causes noticeable visual artifacts.

- There is a problem inherent to image-based techniques authors call cracks. This phenomenon occurs when two close objects are replaced by impostors; if the user changes enough his position the visibility relation between them will change but their impostors will not reflect the change and a hole (the crack) will appear in the final image. Though some authors have presented solutions to alleviate this problem (for instance, see [1] and [20]) there is no method yet that prevents it utterly.

3.4 Weak Visibility Computation

Works in this field are characterized by a redefinition of the concept of visibility that is weaker than the original one (hence the term weak visibility). Their goal is to reduce the complexity of the visual space partition to an affordable size. If this redefinition is done properly, the sets of visible polygons in the new partition will be supersets of the sets in the exact visibility partition. In the bibliography, this kind of superset is often called a Potentially Visible Set or PVS for short. As all the elements in a PVS are visible but the contrary does not usually hold, many authors employ the term conservative visibility.

Some authors consider hierarchical view frustum culling as a predecessor of weak visibility computation. Indeed, we can say that view frustum culling algorithms use the following visibility definition: a polygon (or object) is visible iff it is inside the view frustum. In order to discard invisible polygons quickly they use hierarchical data structures to store the scene: Clark in [5] a hierarchy of bounding volumes and Garlick, Baum and Winget an octree in [11].

Besides the works described fully below, there are two that are restricted to the 2.5D case. The first one, done by Teller and Sequin in [22], is restricted to architectural interiors (just one floor) with axis aligned walls. It uses a BSP to partition the scene and presents several algorithms to compute visibility between rooms. The second one, by Yagel and Ray (see [23]), employs a uniform grid whose cells are labeled as full, partially full and empty. It computes visibility between the cells.

3.4.1 Coorg and Teller (1996)

To reduce the visibility space partition's complexity, Coorg and Teller simplify the visibility concept in two ways. In the first place, they constrain themselves to convex objects, be those polygons or polyhedra. In the second place, occlusion is computed individually on a occluder by occluder basis, so an object that is fully occluded by combined action of several occluders, but not by either of them individually will be classified as visible.

These constraints should not be too limiting for several types of scenes. For instance, indoor scenes and city models are particularly well suited because of walls, which are usually convex and are responsible for the vast majority of real occlusions. On the other hand, we can expect that outdoor scenes will show poorer performance because the occlusion is due to combined action of small polyhedra (forests are a classical example).

Anyway, these restrictions pay because they severely diminish the number of kinds of visual events. As a matter of fact, we just have to consider vertex-edge events. Even better, VE events only generate planar visibility boundaries, so we can forget about quadric surfaces.

A naïve visibility algorithm can now be exposed. As a preprocess, the computer calculates all the planes generated by every scene vertex and every scene edge. Then, it associates to each cell of the induced space partition the set of its visible polygons. At run-time, the computer first locates the initial cell where the viewpoint is situated; as the user moves thru, the algorithm checks the current cell's boundary planes for visibility changes.

Unfortunately, this partition is still too large: in a scene of n vertices there are $\Theta(n^2)$ planes and $\Theta(n^6)$ cells, so further simplification is required. In order to do so, we will introduce the concepts of supporting and separating planes between two convex polyhedra. Planes generated by an edge of one polyhedron and a vertex of the other are called separating if each polyhedron is in a different side of the plane, and supporting if both polyhedron are in the same semi-space. They are oriented towards the occluder. Also, we will say that a point satisfies a plane if it is in the plane's positive semi-space.

Say both polyhedra are completely visible from the viewpoint. Then only the separating planes are relevant to track visibility changes, because the only event that can occur is one polyhedron to become partially occluded. On the other hand, if one polyhedron is fully occluded only supporting planes are relevant to detect that the occludee has become partially visible, for partial visibility takes place when the viewpoint satisfies all separating planes, but not all supporting ones.

This set of planes is enough to determine visibility between two convex polyhedra from a fixed viewpoint. But as our goal is to track visibility as the viewpoint moves we will need additional information. This is so because this set changes as the viewpoint moves. Supporting and separating change when either silhouette changes, for they always correspond to silhouette edges. The planes of faces adjacent to current silhouette edges track silhouette changes, so we will have to add them to our set.

Finally, when the occludee is partially visible, we may want to know which polygons are hidden, which are completely visible and which are neither. This implies adding more planes to the set. In fact, we will need planes corresponding to some non silhouette vertices of the occludee; concretely, all the $[VE]$ planes such vertex V is adjacent to an occludee edge that crosses an occluder silhouette edge E in image-space. As happened before, viewpoint motion can modify this subset, so we will need more information to track when edges begin or cease to overlap in image-space. Indeed, we need all the $[VE]$ planes such V is an occluder silhouette vertex that is "inside" an occludee face F in image-space, and E is an edge adjacent to F.

Now, we can define the the relevant plane's set between two convex polyhedra A (the occluder) and B (the occludee) from a fixed viewpoint as

1. The planes of faces of either polyhedra that have a silhouette edge.

2. For each silhouette edge E of A
   (a) Supporting and separating planes containing E.
   (b) All the planes formed by E and a vertex V such as V is in a B edge that overlaps E. (*)
3. For each silhouette vertex V of A
   (a) Supporting and separating planes containing E.
   (b) For each face F of B such V is inside F when viewed from the viewpoint, all the planes formed by V and each of the edges of F. (*)

Star-marked planes are only necessary if we want to compute visibility on a polygon basis. Coorg and Teller demonstrate in [6] that the planes in this set suffice to detect all visual events when the viewpoint moves - that is to say, the set contains all the planes that form the viewpoint's visibility cell. Thus, we could dynamically compute visibility between two convex polyhedra by calculating the relevant plane set at the initial viewpoint and its corresponding set of visible polygons, and maintaining both of them accordingly as the user moves thru. When a visual event takes place, that is, when the viewpoint crosses a relevant plane, the computer has to update the set of relevant planes or the set of visible polygons or both, depending on the nature of the visual event. Let n, m be the polyhedra's sizes; the number of relevant planes is $\Theta(n+m)$, which is also the time complexity of the dynamic algorithm, for an event that causes the silhouette to change can translate into $\Theta(n+m)$ changes of the relevant plane set. This is an improvement over the naïve approach, that has to deal with $\Theta(nm)$ planes, and therefore has also $\Theta(nm)$ worst case time complexity.

Nevertheless, this algorithm is still too costly when complex scenes are considered, because it will need all the $\Theta(n^2)$ pairwise visibility relations. The authors cut the number of relevant plane sets by using a spatial subdivision structure (a kd-tree or an octree) for occludees and by bounding the number of occluders used at each viewpoint.

The implementation described in [6] is restricted to polygon occluders. Due to the axis-oriented nature of nodes of octrees and kd-trees and to the easiness of computing a polygon's silhouette, maintenance of the relevant plane set becomes rather simpler. In fact, the only planes that are necessary are the supporting and separating ones. So, a navigation algorithm can now be sketched:

- At preprocessing stage, compute an object octree or kd-tree.
- At execution time, and for each viewpoint
  - Select k occluders based on their approximate area on the image, where k is a user-given constant (the paper does not give any clue about how this selection is actually made).
  - For each occluder, keep the relevant plane set of some nodes. For wholly visible nodes, only supporting planes are necessary; for wholly invisible nodes, only separating ones. Finally, for partially visible nodes, both supporting and separating planes are relevant. Of course, nodes wholly visible or wholly invisible whose parents are not partially visible need not be taken into account.

  - When the viewpoint changes, test all the relevant planes to know whether it has crossed any of them.

Finally, the authors introduce two possible refinements in order to reduce the time spent in checking plane crossing. The first one, to keep a subset of planes corresponding to those planes in the relevant plane set that intersect an r radius sphere centred at the viewpoint. As long as the viewpoint remains inside the sphere, only the planes of the subset need to be checked. Of course, if the user exits the sphere, the subset has to be recomputed. The radius parameter is not a constant, but is computed dynamically, though the paper does not give any details about how it is actually done.

The second alternative uses an octree that contains the current viewpoint and whose nodes are associated with the relevant planes that intersect them. Besides, the computer keeps the sequence of nodes that contain the current viewpoint; in other words, the sequence of parents of the leaf where the viewpoint is located. When the viewpoint exits the leaf node, the nearest common ancestor between the old leaf node and the new one is computed, and a new sequence of nodes is generated by climbing down the tree. As for the octree dimension, the authors state that the best choice is a root node big enough to intersect all the relevant planes.

Be that as it may, empirical results in the paper do not show any improvement when this octree is used; on the contrary, the time needed to compute visibility changes increases, due to the overhead of the octree structure. As for the sphere, author's tables show slight improvement over the original algorithm if the user moves slowly, and a speed diminishment if not.

Advantages

- Empiric results show significant cullings: about 68% into a building's interior and 36% when walking across a city model.
- It takes advantage of the hierarchical nature of the kd-tree to cull at once large portions of the scene.

Drawbacks

- The dynamic nature of the algorithm restricts the maximum number of occluders that can be employed.
- The algorithm computes all visibility changes across the segment between the old and new viewpoint. As the user moves faster, this segment increases its length, so much computation time is wasted.
- It would not work well in scenes when occlusion is mainly due to many tiny, not individually significant polygons (leaves in a forest, for instance).
- The parameters of the algorithm are not user-intuitive. They hardly relate to the amount of desired culling.

3.4.2 Coorg and Teller (1997)

The 1997 paper of Coorg and Teller ([7]) is a variation of their previous 1996 work. The common points between the two papers are the kd-tree used to group the occludees, the dynamic nature of the algorithm, the use of a set of relevant planes to classify tree nodes according to their visibility properties and the dynamic selection of occluders. However, some differences arise.

- In the first place, this algorithm uses a subset of the relevant planes' set employed by its predecessor. Instead of computing all the separating and supporting planes between occluder and occludee, it only takes into consideration those of them that are generated by and occluder edge and an occludee vertex. Of course, this strategy reduces the size of the set. The price is the loss of some visibility information. Indeed, this algorithm provides exact full visibility detection but only conservative partial visibility detection.

- In the second place, visibility relationship is computed on a polyhedron-versus-node basis. This is a significant improvement over the previous algorithm whose occluders were restricted to polygonal nature. As a matter of fact, the authors give an algorithm to detect joint occlusion by a set of edge connected polygons $A_1, \ldots, A_k$ whose final silhouette is convex. This set jointly occludes B if:

    1. Every polygon A_i of the set partially occludes B, and none occludes it fully.
    2. If A_i and A_j share and edge e, then they lie on opposite sides of e when viewed from the viewpoint.
    3. The viewpoint is in the positive semi-space of all the planes except supporting ones of common edges.

- Next, this algorithm does not maintain the relevant planes' set. When the viewpoint moves thru, it does not compute the changes in the set by checking which planes have been crossed. Instead, it re-computes the set. It does take some profit from temporal coherence, though, as before re-computing the set it tests actual planes for validity. This is possible because every visited node (that is to say, every node whose ancestor is neither fully visible nor fully invisible) has a cache with its supporting and separating planes.

- The authors introduce a more efficient algorithm to compute supporting and separating planes.

- In this paper the authors detail the dynamic occluder selection. They measure polygon occlusion potential with the formula

  $$\frac{A(\vec{N} \cdot \vec{V})}{d^2}$$

  where A stands for the polygon's area, $\vec{N}$ for the polygon's normal, $\vec{V}$ for the viewing direction and d for the distance from the viewpoint to the centre of the polygon. It is straightforward that this metric benefits large, near polygons that face the user. It is unclear in the paper, though, how does this polygon-based selection procedure relate with the polyhedral nature of occluders.

Anyway, in a preprocessing step the computer traverses all kd-tree leaves and samples all possible viewing directions. For each sample, it selects the best k occluders using the above metric and associates them to the leaf and the sampled viewing direction (k is a user-defined constant). At run time, the algorithm selects the leaf where the user is located and the sampled viewing direction closer to the current one to obtain a set of occluders.

- Finally, the paper introduces a technique to get rid of small detail objects, even when they are not occluded by the big occluders close to the user. At preprocessing time, the computer calculates a set of potential occluders for each detail object: the objects in this set will be those which are large when viewed from the centre of the detail object.

  When the user is navigating through the scene, the algorithm uses these sets to check for occlusion of detail objects that have not been culled by the main procedure. However, it is obscure in the paper how does the computer decide which objects are detail ones, how many detail objects are in the scene and how many occluders should be associated to them. These are important questions because wrong guesses will result in too much overhead or in no speed gain.

Advantages

- The authors show really good figures in the scenes tested. View-frustum culling and occlusion culling purged most of scenes' polygons: 96.4% and 97.4% for a city and a building's interior.
- The algorithm takes profit of spatial coherence by means of the kd-tree. In fact, the author's tests show that the computer just tests about a third of the kd-tree nodes.

Drawbacks

- The dynamic nature of the algorithm restricts the maximum number of occluders that can be employed.
- It would not work well in scenes when occlusion is mainly due to many tiny, not individually significant polygons.
- The parameters of the algorithm are not user-intuitive. They hardly relate to the amount of desired culling.

3.4.3 Hudson, Manocha et al.

The algorithm in [14] is, again, structured into a pre-processing stage and an on-line stage. Before navigation, the computer selects potentially good occluders and arranges them into a voxel space. At navigation time, the algorithm first builds shadow frusta for some of the occluders associated to the current viewpoint's voxel, and then culls the scene with them. As we want this culling to be as fast as possible, the scene is structured into an axis-aligned bounding box hierarchy.

The pre-process first selects all convex objects in the scene. Then, and for every one of them, the computer estimates the region where its solid angle is bigger than a fixed threshold.[2] Were the occluder an ellipse, this region would be an ellipsoid; if a sphere, a concentric sphere. To simplify calculations, the computer approximates every occluder by a sphere or an ellipse. Anyway, once this region is computed, its occluder is associated to every voxel that intersects the region.

Besides, the authors describe another occluder goodness measure, though they do not employ it in their tests. For each occluder and region, it would be useful to estimate the amount of geometry that will be occluded. In order to do so, the computer could sample some viewpoints in the voxel, build their shadow frusta and count how many objects were shadowed by the occluder. The average of the scene's fraction that is actually occluded would serve as an estimation of the occluder's goodness, instead of the solid angle.

Be that as it may, when navigation takes place the algorithm first filters the current list of potential occluders with the view-frustum and then sorts it using the solid angle metric. Only the first k occluders will be used to cull the scene.

Finally, the computer traverses the scene's hierarchy tree and tests each node against each of the $k+1$ shadows frusta (it does view-frustum culling and occluder-culling simultaneously). Every node that is completely outside each of the shadows will be rendered. On the contrary, nodes that are completely inside any of the shadows will be discarded. Finally, partially contained nodes will result in recursive classification of their sons. As this test's speed is crucial to the overall performance of the algorithm, the authors introduce a new algorithm to classify axis-aligned and arbitrary-aligned bounding boxes against a shadow frustum. Though not asymptotically better than previous ones, authors' tests show significant improvement.

Advantages

- Presents a new occluder selection algorithm and an also new, faster algorithm to purge most of the current hierarchical data structures with a shadow from an occluder (octrees,kd-trees, axis-aligned or not hierarchical bounding boxes, etc.).
- Good empiric results. The author's report culling of about 40% of the scene.
- It uses a specialized data structure to store occluders.

---

[2] In fact, the algorithm uses Coorg and Teller's estimation of the solid angle (see [7])

Drawbacks

- The new occluder selection algorithm has not been used in large models. Tests were performed using only Coorg and Teller's metric ([7]).
- As the computer does occlusion on-line, it cannot take into consideration many occluders (the authors, though, state that their tests show that no significant improvement would be achieved were the number of occluders be increased).

3.5 Miscellaneous work

Last, we will sketch some papers that do not fall in any of the three former categories. They have some common points, though. Mainly, that they all work in image space and that are dynamic.

3.5.1 Hierarchical z-buffer

The work of Greene, Kass and Miller in [13] has several impressing features, but is tampered with its inability to make profit of current graphics hardware.

As a preprocessing phase, the algorithm builds an object octree. Each node has a list of the objects that: (a) are completely inside the node's associated cube, and (b) are not completely contained in either of its eight sons. The number of primitives that are completely inside the node acts as the subdivision criterion.

This octree alone would allow significant rendering speed improvement if z-buffer hardware were able to quickly answer the question 'Would this polygon be visible if rendered?' Indeed, instead of rendering by traversing the list of scene objects, we could traverse the object octree and query the z-buffer for each node's cube visibility. A negative answer would mean that we could safely skip all the node's associated geometry, as well as its descendants'. On the other hand, an affirmative answer will result in rendering of the node's geometry and in recursive handling of its sons. The algorithm uses the octree to rapidly reject significant portions of the scene.

Furthermore, we could increase rendering speed if we were able to decide quickly about a node's visibility. In order to do so, the authors introduce a new data structure they call the z-pyramid. The algorithm builds it from the classical z-buffer in a recursive manner: every z-value of a i level buffer is computed as the maximum of the z-values of a 4×4 block at $i + 1$ level. As a final improvement, when inquiring about polygon visibility the algorithm does not scan-convert the polygon and test each pixel separately; instead, it calculates the polygon's minimum z-value.

Finally, the algorithm exploits temporal coherence -that is to say, the fact that visibility does not usually change abruptly from frame to frame- by maintaining a list of previous frame visible nodes. This list is used to initialize the z-pyramid and frame buffer before the octree-based rendering begins; besides, all the nodes involved are marked as drawn and will be ignored when traversing the octree. This procedure will reduce the time the computer spends in building the z-pyramid, for most of the current frame visible geometry will have been drawn already and negative answers are faster than affirmative ones.

Advantages

- Affordable memory requirements.
- Robustness, as in almost all image-space algorithms.
- Null scene structure requirements.
- The algorithm takes full profit of object-space spatial coherence, image-space spatial coherence and time-coherence.

Drawbacks

- The fact that current graphics hardware does not allow for quick z-buffer queries implies that polygon rendering and z-buffer have to be software implemented.

3.5.2 Octree-based volume rendering

Chamberlain, DeRose, Lischinski, Salesin and Snyder present in [4] a simple and easily implementable technique. It uses an object octree to structure the scene and associates six pairs (color, opacity) to every node, one pair for each of the six faces of the node's corresponding cube. The first component is the color that would be seen by an observer situated far away. They compute it by rendering the node's geometry at a coarse resolution (rather coarse, indeed, for they use a 4×4 bitmap) and averaging the image. The second component stands for the emptiness ratio of the node, calculated as the fraction of the node's face that is covered by its geometry. The algorithm builds the octree as a preprocessing step. The subdivision criterion is the cost of rendering the primitives (approximated by the number of triangles required) measured against the cost of rendering the cubes of its children.

The dynamic phase is rather simple. It traverses the tree in a back-to-front fashion. If a node's projection in the final image is bigger than a fixed threshold (say, one pixel), then its raw geometry is rendered. Otherwise, the node's cube is used instead.

Advantages

- The algorithm's implementation is quite simple and straightforward.
- As almost every computation is done in space-image, the implementation will be robust.

Drawbacks

- The authors themselves limit the usefulness of their method to scenes with "suspension-like distributions of primitives that are uncorrelated and small relative to the leaf cells of the octree." Otherwise, noticeable visual artifacts appear.

3.5.3 Hierarchical Occlusion Maps

Zhang, Manocha, Hudson and Hoff introduce in [24] a new approach to compute visibility. Suppose an oracle is provided so for each point of view we have a set of potential occluders. Then, they split the problem of knowing if a polygon is occluded by such set in two simpler questions: Will the polygon wholly collide with the occluders when rastered? And, if so, is it actually behind the occluders? If answers to these two questions are both affirmative we can be positive that the polygon will not be visible, so we do not have to render it.

Of course, this line of action would only be practical if we were able to solve both sub-problems in less time that the hardware takes to render the polygon by z-buffering. In order to do so, the authors introduce two new data structures they call hierarchical occlusion map (HOM hereafter) and depth estimation buffer.

Say the oracle has provided us with a set C of potential occluders. A naïve way of testing whether a polygon P intersects C in image space follows: first, render all C polygons in white on a black bitmap; second, simulate P's rendering. If any of P's pixels was black then P cannot be fully occluded by C. In this scheme, the bitmap plays the role of a opacity matrix. It goes without saying that this method is quite slower than hardware z-buffering, but further improvement can be achieved.

In the first place, as rendering a 2D isothetic box is quite faster than rendering an arbitrary 3D polygon, we will just consider the 2D bounding box of P's projected 3D bounding box. We can do still better if we employ some sort of tree-like structure similar to that in the hierarchical z-buffer in [13]. Given an occlusion map of our selected occluders, we can easily construct a coarser, smaller one, by block-averaging the former, for instance in 2×2 pixel blocks. Moreover, we can iterate this process so to obtain a hierarchy of occlusion maps. Then, we can accelerate the overlap test in the following manner: first, a straightforward computation will let us know which level of the hierarchy has a pixel size (almost) equal to the box size; then, beginning at this level, the algorithm can check each one of the overlapping pixels not to be opaque. If all of them are opaque, we can assert P fully overlaps with our occluder set. Otherwise, the algorithm recursively descends the hierarchy and tests all the corresponding sub-pixels. As a plus, this approach allows the user to do approximate culling, just by changing what the algorithm takes as "opaque": instead of requiring opacity to be 1, we can consider that opaqueness starts at smaller values, as, say, 0.95. This means that little holes in occluders will not be taken into account when checking for occlusions.

As for the depth test, the authors offer two solutions. The simpler one is to compute the maximum z-coordinate of the occluder set. This defines a z-plane parallel to the near plane. An object is taken as occluded iff it overlaps and it is beyond this z-plane. The second solution refines the former by computing the maximum z-coordinate of each occluder in the set. The depth estimation buffer is a z-buffer where the bounding rectangle of each occluder's bounding box is rendered with z-value the occluder's maximum z-coordinate. Once constructed, we can conservatively test if a polygon P is beyond the set of occluders by, again, rendering the bounding rectangle of its bounding box on the depth estimation buffer.

Now, the only pending problem is the occluder oracle. As a preprocessing step, the algorithm filters the scene rejecting objects it thinks are too small, and constructs a bounding box hierarchy with the remaining ones. Besides, in order to perform view frustum culling more efficiently, it also builds a bounding box hierarchy of the whole scene. Neither of these two preprocessing steps takes significant time. At execution time, the algorithm first culls both scene and occluder databases using the view frustum. Then, iteratively selects the nearest occluder till a fixed threshold of occluder polygons is reached, so the HOM and the depth estimation buffer can be built. Finally, it tests every non-culled object in the scene database for occlusion.

Advantages

- The proposed algorithm has null scene structure requirements, for it is able to work with polygon soups and all kind of degeneracies.
- Also due to its image-space nature, the algorithm is quite robust.
- When computing occlusion, occluders are taken as a whole and not individually, so much more scene pruning is expected.

Drawbacks

- In order to achieve good frame-rates, neither the depth estimation buffer nor the HOM are sampled at the same resolution as the final image. In fact, the authors' tests were done at resolutions of 1024×1024 for the final image, 256×256 for the maximum-resolution level of the HOM and 64×64 for the depth estimation buffer. This difference in resolutions causes aliasing problems.
- As the authors note, approximate culling can result in visual artifacts. No matter how small is an occluder's hole, the mix of a dark occluder and a bright occludee will lead to noticeable popping when the users zooms in and out.
- Nevertheless, the strongest objection to the algorithm deals with occluder selection. Though the paper shows tests on three different scenes (a city model, a dynamic model and a submarine machine room), the occluder selection algorithm was manually tuned in each case. While in the first two models the occluder database was identical to the scene database, in the third one not only the algorithm pruned the smallest objects but

also applied simplification algorithms to occluders. At execution time, the HOM was not constructed using the occluders' originals models, but their simplified versions, so to decrease HOM's computation time. However, there are currently no simplification techniques that ensure that the simplified result is completely inside the original object, and therefore this approach will cause errors in the visibility computation.

3.6 Summary

There are three different approaches to the problem of navigating thru very complex, polygonal scenes. Multiresolution techniques create simplified polygonal impostors of each scene's object. The computer can use them instead of originals for objects whose pixel contribution are low enough. Their main disadvantage lies in their object-oriented nature. Indeed, these algorithms render all objects in the scene, so there is an upper bound on the amount of geometry they can save the computer from render, because the number of rendered objects remains constant. There are some multiresolution algorithms that are able to cope with scenes as a whole, but their results are yet not visually satisfactory enough (see [3]).

On the other hand, image-based rendering techniques are able to melt big clusters of objects into a simpler object that can be rendered quickly and that represents the cluster faithfully enough. Unfortunately, they are burdened with visual artifacts and huge memory requirements.

Algorithms in the weak visibility computation family perform occlusion culling while navigating. This cull can be done quickly enough if the definition of visibility is weakened properly and scenes are stored hierarchically. If the scene is densely occluded and occlusions are caused by the isolated action of some objects then current algorithms are able to throw away significant parts of the scene. But they perform poorly when occlusion is due to the joint action of many small objects.

Finally, all the works exposed share their inability to be intuitively parametrized. In fact, many of them have input parameters to tune them, but, as the relationship between the parameters and the final frame-rate is utterly obscure, the tuning process becomes a trial-and-error procedure. This behaviour is particularly annoying because it has to be suffered again and again for every different scene.

4 Our proposal

We have chose weak visibility to be our area of research. We think that there is still room for improvement in this field. Besides, there is a wide range of applications were we feel weak visibility is the best suited approach. For example, architectural and ship-design environments, and, more generally, whatever scene that, though populated by hundreds of thousand of polygons, is so densely occluded that the number of visible polygons from every fixed viewpoint is much lower; so low indeed that, were a visibility oracle be provided, nowadays graphic

	Object or Image Space	Scene Data Structure	Static or Dynamic	Impostors	Occluder Data Structure	Scene Specific
CT:96	O	kd-tree	S+D	No	?	Interiors
CT:97	O	kd-tree	S+D	No	scene kd-tree	Interiors
HMC+:97	O	HBB	S+D	No	Voxels	Interiors
GKM:93	I	Octree	D	No	-	No
CDL+:96	I	Octree	D	Yes	-	Yes
ZMH+:97	I	HBB	D	No	HBB	No
MS:95	I	Octree	S	Yes	-	No
Sch:95	I	None	D	Yes	-	No
SS:96	I	kd-tree	D	Yes	-	No
Ali:96	I	Voxels	D	Yes	-	No
SDB:97	I	Graph	S	Yes	-	Cities
AL:97	I	Graph	D	Yes	-	Rooms

Table 2: Summarizing table. HBB stands for Hierarchy of Bounding Boxes

workstations could be able to render it at interactive frame-rates. Sadly, we know that such an oracle is beyond the computation power of today's hardware, and will be for some time still. Thus, we will have to restrict ourselves to non-exact oracles whose answers (i.e., sets of visible polygons) are oversized.

But it does not really matter how big the set of visible polygons is as long as its size is within our hardware's capabilities. As a matter of fact, a common problem of all the algorithms to our knowledge of either of the three families is that they are not easily parametrized, and in several cases cannot be parametrized at all. There is no way the user can relate the user-defined variables to his desired frame rate, and the only procedure at his disposal is the trial-and-error one.

Of course, the solution would be an algorithm that asked the user for his desired maximum number of polygons per frame and tried to ensure that every possible set it provided would be within that limit. This behaviour will allow the user not only a better control over the final result, but could also be the way to navigate very complex scenes in a wider range of computers. The computation of the visibility data structure could be done in an expensive, state-of-the-art machine but parametrized for another, cheaper, not so sophisticated computer, provided the latter had enough memory to keep the visibility data. It goes without saying that this behaviour cannot be utterly guaranteed even if a perfect oracle were at our disposal. Nevertheless, an algorithm that took this into account could spend most of its time into very complex zones, and less into simpler ones.

Due to the complexity of the visibility spatial partition and to the fact that we want fast access to our data structure, we could think of employing some kind of hierarchical structure like octrees or kd-trees to store visibility information. This kind of data structures have proved to be very efficient in many areas of computing science. But in order to use an octree to compute and store this kind of information in an efficient manner some requirements must be fulfilled

i) We must be able to compute visibility inside a node at a reasonable speed.

ii) The visibility information of a node should help the computation of its sons.

iii) Finally, visibility must be conservative if we want our octree not to grow beyond current hardware's memory capabilities.

Before we can give answers to this questions we have to introduce some formal definitions and properties.

Definition 1 *The shadow from a point p of a set $A \subseteq \mathcal{R}^n$ is*

$$S(p, A) = \{q \in \mathcal{R}^n \mid \overline{pq} \bigcap A \neq \emptyset \ \wedge q \notin A\}$$

where $\overline{pq}$ is the segment between p and q

It follows naturally the concept of shadow from a set of points.

Definition 2 *The shadow from a set $P \subseteq \mathcal{R}^n$ of a set $A \subseteq \mathcal{R}^n$ is*

$$\begin{aligned} S(P, A) &= \{q \in \mathcal{R}^n \mid \forall p \in P : \overline{pq} \bigcap A \neq \emptyset \ \wedge q \notin A\} \\ &= \bigcap_{p \in P} S(p, A) \end{aligned}$$

If we restrict ourselves to convex occluders, a quite interesting property arises: there are some observer sets whose shadow can be exactly computed from the shadows of a few points of the set. In other words, we can compute shadows by sampling. This property of shadows was first noticed by Nishita, Okamura and Nakamae in [16], though they did not prove it.

Lemma 1 *If $A \subseteq \mathcal{R}^3$ is a convex set and there is a hyperplane that separates A from the segment $s = \overline{p_1 p_2}$, then*

$$S(s, A) = S(p_1, A) \bigcap S(p_2, A) \tag{1}$$

Proof. If $x \in S(s, A)$, it follows immediately from the definition that x will also be in both $S(p_1, A)$ and $S(p_2, A)$. So, we have to prove that if x is in $S(p_1, A)$ and in $S(p_2, A)$, then it will also be in $S(s, A)$.

If p_1, p_2, x are collinear it holds trivially that $\forall p \in s : x \in S(p, A)$, and the proof is over. If they are not, say Π is their supporting plane. From now on we can restrict ourselves to $\mathcal{R}^2$, for $A' = A \bigcap \Pi$ is also convex.

Now, consider two points $x_i \in A' \bigcap \overline{xp_i}$ (see figure 1); their existence is guaranteed by the fact that x is in $S(p_1, A)$ and in $S(p_2, A)$. As A' is convex, the segment $t = \overline{x_1 x_2}$ lies completely inside of A'. Moreover, as x_1 and x_2 are in different edges of the triangle xp_1p_2, the supporting line of t separates x from s. Therefore, say p is a point of s; as the segment $\overline{px}$ is completely inside the triangle and intersects t at some point q, we can assure that $x \in S(p, A)$, so the proof is complete. $\square$

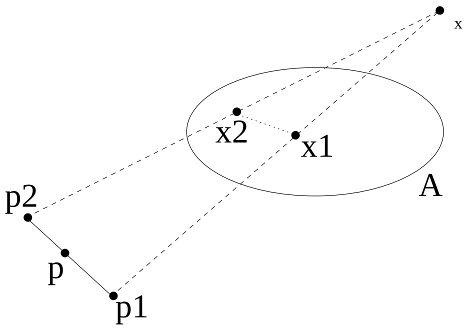

Figure 1: Computing the shadow from a segment

Lemma 2 *If $A \subseteq \mathcal{R}^3$ is a convex set and there is a hyperplane that separates A from the polyline $P_n = [p_1, \ldots, p_n]$, then*

$$S(P_n, A) = \bigcap_{i=1}^{n} S(p_i, A) \tag{2}$$

Proof. We will prove it by induction on the number of points of the polyline. If $n > 2$, then

$$\begin{aligned} S(P_n, A) &= S(P_{n-1}, A) \bigcap S(\overline{p_{n-1}p_n}, A) \\ &= \left(\bigcap_{i=1}^{n-1} S(p_i, A) \right) \bigcap S(p_{n-1}, A) \bigcap S(p_n, A) \\ &= \bigcap_{i=1}^{n} S(p_i, A) \end{aligned}$$

□

Now we are ready to introduce a fundamental theorem that will allow us to compute visibility between a node and a single convex occluder taking into consideration the shadows of the node's vertices only.

Theorem 1 *If $A \subseteq \mathcal{R}^3$ is a convex set and there is a plane that separates A from the closed and bounded polyhedron $C \subseteq \mathcal{R}^3$, then*

$$S(C, A) = \bigcap_{i=1}^{n} S(p_i, A) \tag{3}$$

where p_i are the n vertices of polyhedron C.

Proof. We just have to prove that $\bigcap_{i=1}^{n} S(p_i, A) \subseteq S(C, A)$, for the other inclusion is trivial. That is to say, we want to see that if x is invisible from every vertex p_i, then it will also be invisible from every point p in C.

As C is closed and bounded, the segment $\overline{px}$ intersects at least one face f of C at some point q (see figure 2), and x is invisible from p iff it is so from q. Say

r is one of the infinite lines on the supporting plane of f that are incident to q. Then, there is a polyline P_n formed by f's edges and segments interior to f such that its two extreme segments e_1 and e_2 intersect r at points q_1, q_2 respectively and such that point q is in the segment $\overline{q_1 q_2}$ (in figure 2, this polyline could be $e_1 s e_2$). Now,

$$x \in \bigcap_{i=1}^{n} S(p_i, A) \overset{(2)}{\Rightarrow} x \in S(P_n, A) \Rightarrow x \in S(e_i, A) \overset{(1)}{\Rightarrow}$$

$$x \in S(q_i, A) \overset{(1)}{\Rightarrow} x \in S(q, A) \Rightarrow x \in S(p, A)$$

and the proof is complete. □

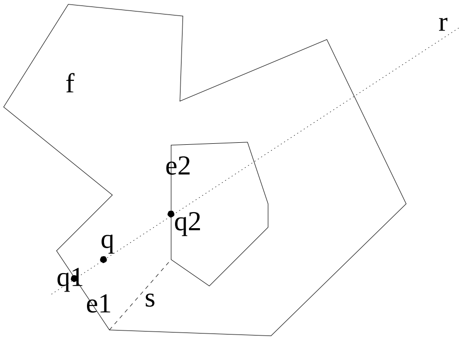

Figure 2: Computing the shadow from a polyhedron

With this results we can now sketch a first version of our algorithm. The computer is provided with an arbitrary polyhedral scene and two parameters: the maximum number of polygons our computer is able to render within our desired frame-rate and the maximum depth of the visibility octree. This octree can be constructed recursively in the following manner

i) For each node's vertex, select some convex occluders. This can be done using the algorithm employed in [7], for instance.

ii) Compute visibility of each node's vertex by computing their shadows (convex truncated pyramids) from each of its occluders and purging the scene with them. Every scene object that is completely inside one of this pyramids is added to the current vertex' set of invisible objects corresponding to the current occluder. The problem of computing the shadow of a convex occluder from a fixed point p can be reduced to the problem of computing the silhouette of the occluder from p. There are several algorithms to solve this problem.

iii) Say I_{ij} are the sets of current node's vertices, where i runs from 1 to 8 and j from 1 to the maximum number of selected occluders, and let N be the set of objects that are inside the current node's cube. Consider now the sets

$$I_j = \bigcap_{i=1}^{8} I_{ij} \setminus N$$
$$I = \bigcup_{j} I_j$$

The set I_j contains all scene objects that are invisible from every viewpoint inside the visibility node due to the single action of the occluder j. And I is the set of objects invisible from every inner viewpoint due to some of the selected occluders. We associate this set to the current visibility octree node.

iv) If the number of polygons in this set is greater than the given threshold and we have not yet reached the maximum tree depth, the algorithm splits the node and computes each son's visibility. If not, we associate the set I to the node and are over.

We can do still better if we realize that every time the computer calculates visibility from a point chances are it has computed it several times before. Indeed, every time a node is subdivided the algorithm recomputes visibility of each of its eight vertices when dealing with its sons. We can avoid this annoying behaviour if we associate all the invisibility sets of each vertex to every leaf instead of their intersection. Before computing a new node's vertex visibility from scratch, the computer ensures it has not been computed previously by searching the point in the octree. If the leaf returned by the search procedure is the current leaf, we know that the point's visibility has not been computed before. We can achieve this if we programme carefully the searching method. If the searched point is a non-root vertex node, sooner or later the searching procedure will have to choose between several nodes. We just have to ensure that the selection follows the order in which the constructing procedure computes the sons of a node: the node which would be computed first is the one the search procedure must choose. The intersection and union steps are then performed dynamically.

Step (i) can be enhanced also. The original procedure in [7] had to be careful not to select too many occluders, because an excess of them would result in too much overhead when navigating thru the scene. This is so for two reasons: first, because occlusion computation is done dynamically; second, because the computer cannot ever be positive about the goodness of an occluder and some of them will not purge the scene significantly. But our algorithm performs almost all its computations before user navigation, so it can test much more occluders. Those occluders whose occlusion effects are insignificant will be dismissed. Moreover, it can select them incrementally, starting with a fixed number of occluders and adding new ones while the number of visible polygons is not good enough.

Besides, we can enhance step (ii) severely if we use some kind of hierarchical spatial subdivision data structure to store the scene. We can use an object octree or a kd-tree, or whatever data structure that aids us to intersect our truncated convex pyramids faster with the scene. Moreover, this structure will

#objects/pipe's size	Long pipes	Medium pipes	Small pipes
390 (40 pipes/room)	190/368/245 49/94/63	189/351/242 48/90/62	177/333/288 45/85/74
930 (100 pipes/room)	464/856/577 50/92/62	453/844/567 49/91/61	413/789/551 44/85/59
1830 (200 pipes per room)	847/1665/1154 46/91/63	851/1641/1117 47/90/61	830/1554/1063 45/85/58
9030 (1000 pipes/room)			4022/7568/5210 45/84/58

Table 3: The visibility octree tested on different scenes and different viewpoints. Each cell shows the number and ratio in % of visible objects at three qualitatively different viewpoints. In x/y/z, the first number corresponds to a viewer located at the corner and facing a wall; the second one, to a viewer at the centre facing a door; and the third one, to a viewer at the centre facing a door.

save us memory space because it will reduce the size of the visibility octree if we replace the sets of invisible objects with sets of invisible nodes. Note that after the visibility octree is constructed we do not have to keep this data structure, only the lists of objects in nodes that are referenced in our visibility octree (though, it would probably be advisable to keep it to be able to do hierarchical view frustum culling). Figure 3 shows a first version of the visibility octree computation algorithm.

4.1 First results

In order to check the feasibility of our algorithm we implemented and tested it. At this stage, our main purpose was not to get a fully optimized programme, but rather some prototype that would allow us to test if the amount of memory needed for the visibility octree was affordable or not.

The test was done on a Silicon Graphics Onyx workstation equipped with 128Mb of RAM and two 194MHz R10000 processors, although our implementation did not take any advantage of the second processor. The largest of the scenes we used is shown in figure 4. There are nine rooms and one thousand pipes per room, for a total of 54180 polygons. Every object in the scene is convex. The computer took one day to build its visibility octree with the following parameters: five levels of depth, ten polygons per leaf maximum and a source object octree also five levels deep and with a maximum of one object per leaf. As we chose extremely low maximums both octrees reached maximum depth, so the algorithm was put under considerable stress. During the computation of the visibility octree, the programme reached a memory usage peak of 500Mb. However, after the computation was finished the amount of memory used did not surpass 50Mb. This quantity is rather low, as the code and the scene without the octrees used almost 30Mb.

Once the visibility octree was computed, we requested visibility information at three qualitatively different viewpoints. Table 3 shows the results for the scenes we tested. All the scenes have nine rooms crowded with pipes. We generated different scenes changing the sizes of the pipes and the number of pipes per room. It is noteworthy that results were almost independent of the scene used. The first viewpoint corresponds to an observer at the corner of the scene and near a wall. The computer was able to cull about 50% per cent of the geometry, as was to be expected. The second viewpoint is located at the centre and facing a door. Not surprisingly, the computer could not throw away but small parts of the scene. Finally, the third viewpoint was chosen to be and intermediate between optimal and worst situations. It was located at the centre of the scene, but close to a wall. The computer could get rid of the 40% of the scene. Figure 5 shows the visible portion of the scene as reported by the visibility octree at the first and third viewpoints. It is clean in the pictures that culling could still be improved significantly, even in the optimum situation. This is so because the object octree does not split objects that are in more than one node, and thus an object can be culled away if and only if all the leaves it intersects are completely inside a shadow. We could expect better results should some tighter hierarchical data structure be used.

5 Conclusions and future work

Today's graphics hardware is unable to satisfy current user's requirements. To solve this problem, the computer graphics community has developed many techniques that can be grouped in three families: multiresolution algorithms, image-based rendering and weak visibility computation.

We have introduced a new weak visibility algorithm that we believe has several advantages over previous work. Namely,

i) Occlusions are computed at the preprocessing stage. The computer can increase the number of occluders if the amount of culled geometry is not enough. Previous weak visibility works performed occlusions at execution time, so they had an unavoidable bound on the maximum number of occluders they could use.

ii) The visibility octree has two input parameters, maximum depth and maximum number of polygons per leaf. Both are directly related to the goodness of the result. The former controls the amount of memory used, and the latter allows better control over the frame ratio if the graphics hardware specifications are known -i.e., the maximum number of polygons it is able to render in a second. Of course, this control is not perfect. Even if we computed exact visibility, we could not guarantee that the number of visible polygons at any viewpoint did not exceed the capabilities of the hardware. But there are many environments where the complexity of visible geometry at any fixed viewpoint is low, whereas the total number of polygons is extremely high (indoor scenes, for instance).

iii) Memory requirements are high when the visibility octree is being computed, but reasonably low once the computation is over. Besides, CPU requirements at execution time are almost null. Thus, although the preprocessing is limited to high-end, expensive, state of the art workstations, the dynamic phase is not, so it is possible to preprocess in high-end workstations and navigate in cheaper computers. Moreover, the preprocessing can be done considering the graphics hardware of the target, low-end computer.

ComputeVisibilityOctree (scene:ObjectOctree, occluders:set, node:VisOctreeNode, ***VAR*** *vis:VisOctree)*

```
for $i := 1$ to 8 do
    $obs_i$ :=GetVertexCoordinates($vis, node, i$)
    $I_i$ :=ComputeInvisibilityAtVertex($vis, node, i, scene, occluders$)
end for

$I := \bigcup_{o \in occluders} \bigcap_{i=1}^{8} I_i[o] \setminus$ GetObjectsInsideNode($scene, vis, node$)

$V := \overline{I}$

if NumPolygons($V$) $> MAX\_POLS \wedge$ Depth($node$) $< PROF\_MAX$ then
    SubdivideNode($vis, node$)
    for $i := 1$ to 8 do
        ComputeVisibililityOctree($V, occluders$,Son($vis, node, i$), $vis$)
    end for
else
    Setnode($vis, node, < V, I >$)
end if
```

ComputeInvisibilityAtVertex(scene: ObjectOctree, vis: VisOctree, node: VisOctreeNode, vertex: integer, occluders: set) **returns** *vector of set*

```
$p$ :=GetVertexCoordinates($vis, node, vertex$)
$node'$ := Locate3DPoint($vis, p$)
$HasNotBeenComputedBefore := (node = node')$
if $HasNotBeenComputedBefore$ then
    $I = \emptyset$
    for all $o \in occluders$ do
        $s$ = ComputeShadowPyramid($p, o$)
        $I[o]$ = InnerNodes($scene, s$)
    end for
else
    $< V, I >$:=GetNodeItem($vis, node'$)
end if
return $I$
```

Figure 3: Construction of the visibility octree in algorithmic notation

On the other hand, we are aware there is much work pending. In the first place, the algorithm has not been tested enough, and there is room for several optimizations in the code. In the second place, we feel that the occlusion field has not been fully researched yet. We have scheduled our future work as follows

i) As pointed previously, the object octree is probably not the best hierarchical structure to purge the scene. We think there are other structures (for example, hierarchical bounding boxes) that supply tighter approximations of objects.

ii) The code has to be optimized. Actually, the code that classifies nodes against volume shadows is rather naïve.

iii) The algorithm must be tested with real scenes. Moreover, it would be useful to have statistic measures -mean, maximum and minimum, and variance- of the number of reported visible polygons in all the leaves of the visibility octree, as well as ratios between the number of reported visible polygons and real visible polygons.

iv) Currently, visibility is undefined when the user is located outside the octree. We want to study if the visibility information can be used somehow when the user is outside the scene's bounding box.

v) The necessity of convexity, although shared by all previous work in the weak visibility area, restricts severely the performance of the algorithm. Some authors propose that the polyhedra should be decomposed into convex parts before any processing. We think this approach is not the best one because it increases considerably the size of the scene. Besides, many of the resultant convex polyhedra will be too small to serve as occluders. An algorithm that, given a polyhedron P, computed a family of convex polyhedra $P_1, \ldots, P_n$ such that every polyhedron P_i is bounded by P but did not necessarily have null intersection with the rest of polyhedra in the family would be better suited.

vi) In the same line, it would be interesting to develop some procedure to, given a set S of small occluders, compute a single polyhedron P that does not occlude more than S or perhaps to compute the union or their shadows.

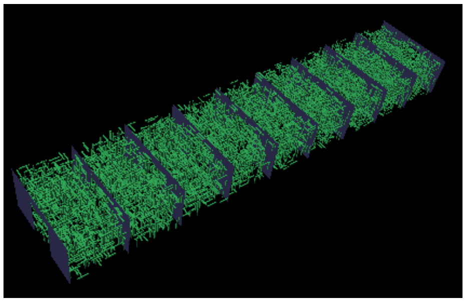

Figure 4: Nine rooms with one thousand pipes each

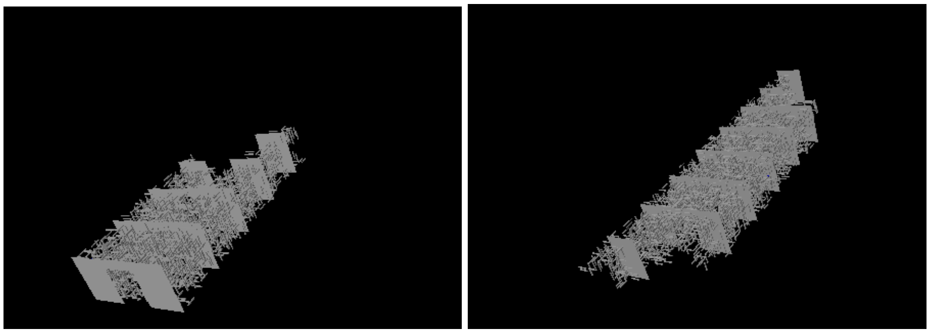

Figure 5: The non-culled geometry at two different viewpoints. From the left corner (left picture) and from the centre and to the left (right picture)

References

[1] Daniel G. Aliaga. Visualization of complex models using dynamic texture-based simplification. In *IEEE Visualization '96*. IEEE, October 1996.

[2] Daniel G. Aliaga and Anselmo A. Lastra. Architectural walkthroughs using portal textures. In *IEEE Visualization '97*, October 1997.

[3] Carlos Andújar. Simplificación de modelos poliédricos. Technical Report LSI-98-1-T, Universitat Politècnica de Catalunya, 1998. [This report is written in Spanish].

[4] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Graphics Interface '96*, pages 132–141, May 1996.

[5] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[6] Satyan Coorg and Seth Teller. Temporally coherent conservative visibility. In *Proceedings of the Twelfth Annual Symposium On Computational Geometry*, pages 78–87, New York, May 1996. ACM Press.

[7] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 83–90. ACM Press, April 1997.

[8] S. E. Dorward. A survey of object-space hidden surface removal. *International Journal of Computational Geometry & Applications*, 4(3):325–362, 1994.

[9] Frédo Durand, George Drettakis, and Claude Puech. The 3D visibility complex: a new approach to the problems of accurate visibility. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 245–256. Eurographics, June 1996.

[10] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 89–100. ACM SIGGRAPH, Addison Wesley, August 1997.

[11] B. Garlick, D. Baum, and J. Winget. Interactive viewing of large geometric data bases using multiprocessor graphics workstations. In *Siggraph '90 Course Notes: Parallel Algorithms and Architectures for 3D Image Generation*, volume 28, pages 239–245, 1990.

[12] Ziv Gigus, John Canny, and Raimund Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542–551, 1991.

[13] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[14] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 1–10, June 1997.

[15] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 95–102. ACM SIGGRAPH, April 1995.

[16] T. Nishita, I. Okamura, and E. Nakamae. Shading models for point and linear sources. *ACM Transactions on Graphics*, 4(2):124–146, April 1985.

[17] H. Plantinga and C. Dyer. Visibility, occlusion, and the aspect graph. *International Journal of Computer Vision*, 5(2):137–160, 1990.

[18] G. Schaufler and W. Stürzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):C227–C236, September 1996.

[19] Gernot Schaufler. Dynamically generated impostors. In D. W. Fellner, editor, *Modeling - virtual worlds - distributed graphics (MVD'95 Workshop)*, pages 129–136, November 1995.

[20] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum*, 16(3):C207–C218, 1997.

[21] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.

[22] Seth J. Teller and Carlo H. Sequin. Visibility preprocessing for interative walkthroughs. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):61–69, July 1991.

[23] Roni Yagel and William Ray. Visibility computation for efficient walkthrough of complex environments. *Presence*, 5(1):1–16, 1996.

[24] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings*, pages 77–88. ACM SIGGRAPH, Addison Wesley, August 1997.