SIACG 2002 - 1st Ibero-American Symposium on Computer Graphics
1-5 July 2002, Guimarães - Portugal

Hoops Fusion: Synthesis of View-dependent Convex Occluders from a Set of Objects

Àlex Ríos
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
alex@lsi.upc.es

Isabel Navazo
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
isabel@lsi.upc.es

Abstract
Visibility determination is a requirement to navigate through complex scenes. Occluder fusion algorithms generate convex occluders that are contained in the umbra cast by a group of objects given an area light. Hoops are non-convex, but view-dependent convex, non-planar closed polylines that can be used to compute occlusion for objects that not necessarily have large interior convex sets. In this paper we present an efficient, robust, and incremental octree-based algorithm to synthesize hoops for a set of objects and to compute the hoopís umbra. Experimental results demonstrate the techniqueís effectiveness to compute occluder fusion of non-convex objects.

Keywords
Visibility Culling, occluder synthesis, occluder fusion, octree representation.

1. INTRODUCTION

Despite the continuous improvement in the bandwidth of graphics hardware, navigation of very complex models remains an open problem of interest to graphics researchers.

Occlusion culling techniques focus on ways to compute conservatively whether a set of objects are hidden by one single object called the *occluder*. Cell visibility algorithms subdivide the navigational space into convex cells and estimate the set of visible objects for each cell by sampling visibility at its corners [Cohen-Orí98] [Saona-Vazquezí99]. For the computational efficiency of cell visibility algorithms, the occluders must usually be convex. Recently, several researchers proposed to use *synthesized* convex shapes which can safely substitute the original non-convex occluders and that can be combined to synthesize a larger occluder [Zhang98], [Law99], [Schaufler00], [Andujar00].

In [Brunet01] *hoops* were proposed as synthesized view-dependent occluders. Hoops are non-planar, non-convex polylines inside the original occluders that are seen convex from a view-cell (see fig. 1). Though concave these hoops retain the main properties of convex occluders regarding visibility: occlusion speed and cell visibility computability. Moreover, hoops can be synthesized from non-convex sets that do not contain any large convex set. In this way the family of objects, that can be tested to compute the potential visible set from a given view-cell, can be increased. It is also shown that in order to obtain a best estimation of the visibility, several hoops (usually less than 5) are needed to substitute a general point set.

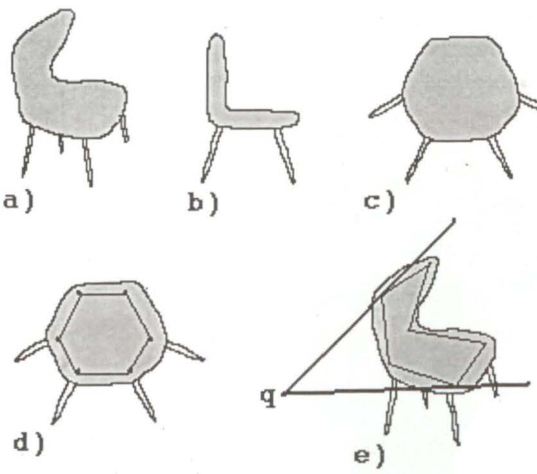


Fig.1: a), b) and c) show a non convex object from different viewpoints. d) shows the hoop of the object seen convex from the viewpoint. In e) the hoop is not seen convex from view-cell q.

In many cases, objects are hidden due to the combination of many, not necessarily convex, occluders. Thus much of the recent research in visibility pre-computation has been focused on combining (fusion) the effect of multiple and arbitrary occluders. *Occluder fusion* algorithms compute larger occluders in the union of the umbra region cast by a set of individual convex occluders. In [Brunet01] it is just stated that hoops satisfy the requirements to be easily fused.

This paper proposes a novel hoop fusion algorithm to compute the integrated occlusion of a set of objects. Each individual object is substituted by its synthesized hoop [Brunet01]. Then the hoop fusion takes place in the umbra of them, and a set of larger hoops are obtained as global occluders. The algorithm is based on a new octree-based technique to obtain the umbra cast by an individual hoop from a view-cell (volume light). These umbra regions are combined with an efficient, robust and incremental algorithm that is also based on an octree representation. Finally the hoops of the integrated umbra regions are computed using the algorithm proposed in [Brunet01].

Section 2 reviews current approaches in occlusion fusion, hoops definition and related algorithms. The basis of our proposed algorithm is presented in section 3. Section 4 describes the implemented hoop fusion algorithm in detail. Finally, sections 5 and 6 show empiric results and discuss conclusions and future work.

2. PREVIOUS WORK

Until very recently visibility computations were limited to occlusions due to a connected convex occluder. That is to say, an object that was occluded by the joined action of two blockers but not by neither of them alone were always classified as visible (see an example in figure 2). Fusion of occluders may be achieved by extending occluders inside previously computed umbra. If an occluder O_2 intersects the umbra U_1 of an occluder O_1, then extending O_2 into U_1 yields a fusioned occluder that culls geometry occluded by O_1 and O_2 together. This procedure can be repeated iteratively with other occluders.

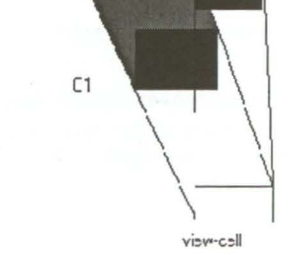


Fig. 2: Object O3 is classified as visible if shadows are treated separately.

In [Koltun00] a technique is proposed that fuses occluders in 2D and 2.5D scenes. Their 2D approach is based on the concept of a virtual occluder, a synthesized convex object that is shielded from a view-cell and thus may be used as an occluder. The basic principle was using the separating and supporting lines between the line-cell (view-cell in 2D) and the current virtual occluder. According to the classification of the new candidate respect those constrains: it is fused and extends that umbra, it is discarded for fusion or it is used to modify some of the constrains. The fusion is done by substituting the current virtual occluders by another behind them that extend the umbra.

[Scahufler00] perform occluder fusion using an octree subdivision and a discretization of the scene and by computing rectangular occluders in the umbra of the original occluders. Then, they fuse these blocks when one intersects the umbra of another one. They use cell-to-object visibility by testing if the discretization of the object bounding box is covered by the accumulated umbra. The most efficient implementation is in 2D and 2.5D scenes and it is constrained to *fat-occluders*.

In a different fashion, [Durand00] project occluders into image space and accomplish occluder fusion by reprojecting on several planes. It is the only cell visibility algorithm that can cope with concave sets although with some limitations. Namely, a plane has to be found whose intersection with the occluder is a single connected surface. Concave occlusion is much slower than convex one.

[Wonka00] method is based on the novel principle that shrinking an occluder by epsilon provides a smaller umbra with an unique property: an object classified as occluded by the shrunk occluder remain occluded with respect to the original occluder when moving the viewpoint no more than epsilon from its original position. For each view-cell a sufficient number of sample points is determined, the visibility is computed for each of them and the occluder fusion is perfomed by shadow aggregation of the umbra cast by the occluders from each sample-point. The implementation is constrained to 2.5D urban scenes and it can be implemented by using hardware graphics cards.

In [Brunet01] is shown that convexity of the umbra from a cell is enough to compute the potential visible set of a cell. Then, by extending the notion of umbra to non-planar polylines, the hoop concept is introduced. Hoops are non-convex, non-planar polylines that cast a shadow that is seen convex from the view-cell (see figure 1). So they can substitute the real occluder to compute conservative (underestimate because they are inside the object) occlusion culling by sampling visibility from the corners of the view-cell. Hoops have a small number of edges and the occlusion culling can be computed in linear time. Efficient tests to check umbra convexity, hoops conditions in linear time and to compute a hoop for a given, non necessarily convex, 3D object are provided. It was shown that for some objects several hoops (usually less than 5) are required to obtain a best occlusion estimation Finally, it is discussed that an algorithm similar to [Schaufler00] and [Koltun00] could be used to compute hoop fusion. The hoop construction algorithm is based on a discretization of the occluder by a classical octree (only black and white terminal nodes).

In the following sections, hoop fusion algorithm is completely developed and justified and some simulations are provided that shows its goodness for synthesizing occluders of a set of objects (region of space).

3. DEFINITIONS AND HOOPS FUSION OVERVIEW

In this section, we will introduce some definitions and the general outline of the algorithm we propose. Given a view-cell C and a set of occluders $\{O_i\}$, the shadows of which intersect, we want to establish a minimal set of hoops. These hoops substitute the set of occluders in order to compute the visibility from C. The underline principle is the following: from the set of occluders $\{O_i\}$, an initial occluder O_1 is selected and its hoop synthesized. The octree of its umbra O_{U1} from C is computed using an efficient and incremental algorithm. Then, a second object O_2 the hoop of which intersects the umbra U_1 , is selected from $\{O_i\}$ and the octree O_{U1} is updated using the hoop of O_2 to represent the union of the the umbras U_1 and U_2. Finally, the hoop of the union is computed using the algorithm presented in [Brunet01]. (See fig. 3).

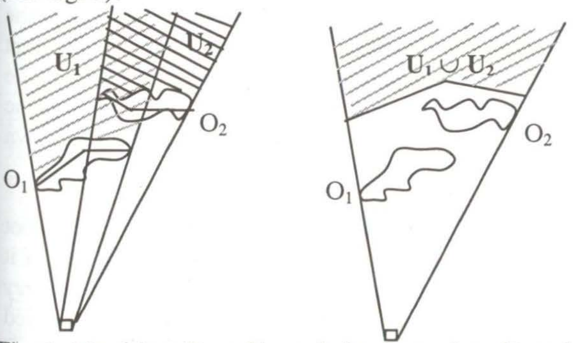


Fig. 3: Note that $U_1 \cup U_2$ occludes more than U_1 and U_2 separately, as explained in section 2.

The following function outline the process.

```
Function fusion(O1,O2:object; C:cell): hoop;
Var
   OU1:octree  //Umbra of the hoop of O1
   OC1, OC2, SO:octree
   h1, h2, h:hoop;
   BB:box;
Fvar
       OC1:=octree(O1); h1:=hoop(OC1, C);
       OC2:=octree(O2); h2:=hoop(OC2, C);
  1    BB:=compute_bounding_box(C, h1, h2);
  2    OU1:=compute_octree_umbra(BB, h1);
  3    SO:=update_octree(OU1, h2);
  4    compress(SO);
       h=hoop(SO, C);
Return h
Ffunction
```

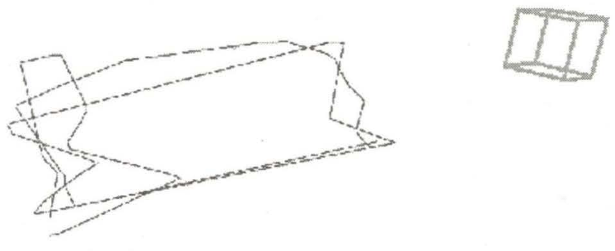
Algorithm 1: hoop fusion of 2 objects.

Given two objects and knowing that O_2 intersects the shadow of O_1 from the view-cell C, the *hoop* function synthesizes a hoop for each object (in fact from the classical octree, *OC1* and *OC2*, of each object). As seen in [Brunet01] every object should be substituted by a set of hoops in order to get a high conservative rate of visibility. Experimentally we have found that the shadows of a set of 5 hoops covers as much as 90% of the shadow of the object. For simplicity, in the following explanations, we consider only one hoop for each object which provides us with the same position as if we were executing the algorithm 1 for all the hoops for each object.

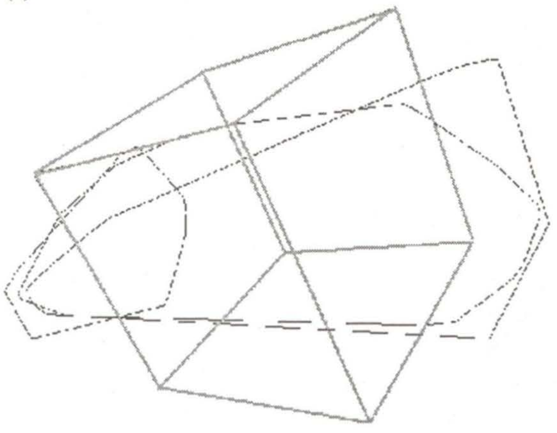
The functions *compute_octree_umbra* and *update_octree* compute the octree of the umbra of the initial hoop and the hoop fusion respectively. The octree of the union of the umbras has black nodes inside the umbras of any of the two objects and white nodes outside. We call this octree the Shadow Octree (SO). Due to the construction algorithm, the SO is over-subdivided, so before use it to compute the returned hoop of the fusion, it should be compressed in order to optimize the hoop construction algorithm.

4. HOOPS FUSION ALGORITHM

In this section we explain in detail the processes involved in our proposed algorithm. Given a view-cell C to compute the octrees O_{U1} and SO, we take advantage of the fact that the hoops and their umbrae are seen convex from the cell (see fig. 1 and 4)



(a)



(b)

Fig 4. (a) Three hoops and the view-cell. (b) The same hoops seen from the view-cell. The umbrae from the cell will also be convex.

We will see that this condition allows us to compute the octree of the umbra incrementally.

To compute the Shadow Octree we follow four basic steps:

1. Compute the bounding box of the SO.
2. Compute the octree of the shadow for the first hoop (O_{U1})
3. Update the octree O_{U1} with the umbra of the second hoop.
4. Compress the Shadow Octree.

4.1 Computing the Bounding Box

The bounding box is needed to find the universe associated to the root of the octree: the region of the space that will contain the shadow region. It is calculated in the following way (see fig 5): first project the shadow of all the selected hoops onto a plane located behind the hoops and the view-cell, then calculate the bounding box that contains the hoops and the projected shadows. The projection plane selected is perpendicular to the line defined by the center point of the view-cell and the center point of the bounding box of the two initial hoops (fig. 5.a and 5.b). The projected shadow is computed by obtaining the intersection of the separating planes defined by the hoop and the view-cell (see next section) with the projection plane (fig 5.c). The final bounding box contains the hoops and their projected shadows (fig 5.d). This process is done before computing any octree.

4.2 Octree of the Shadow for the First Hoop

To compute the shadow octree for the first hoop we take advantage of two facts: that the hoop is seen convex from the cell and that the cell is also convex. It is possible, then to define the separating planes of the cell and the hoop: planes that leave the hoop and the cell at the same half-space and are defined by a vertex of the cell and two consecutive vertices of the hoop. The shadow of the hoop from the cell is a cone bounded by the separating planes of the cell and the hoop. For each edge of the hoop there is at least one vertex of the cell that define a separating plane. The set of separating planes can be computed by a linear test.

Previous classification	Current classification	Final Classification
WHITE	OUTSIDE	WHITE
WHITE	INSIDE	WHITE
BLACK	OUTSIDE	WHITE
BLACK	INSIDE	BLACK
-	TRAVERSED	GREY

Table 1: Classification of the nodes for the first hoop.

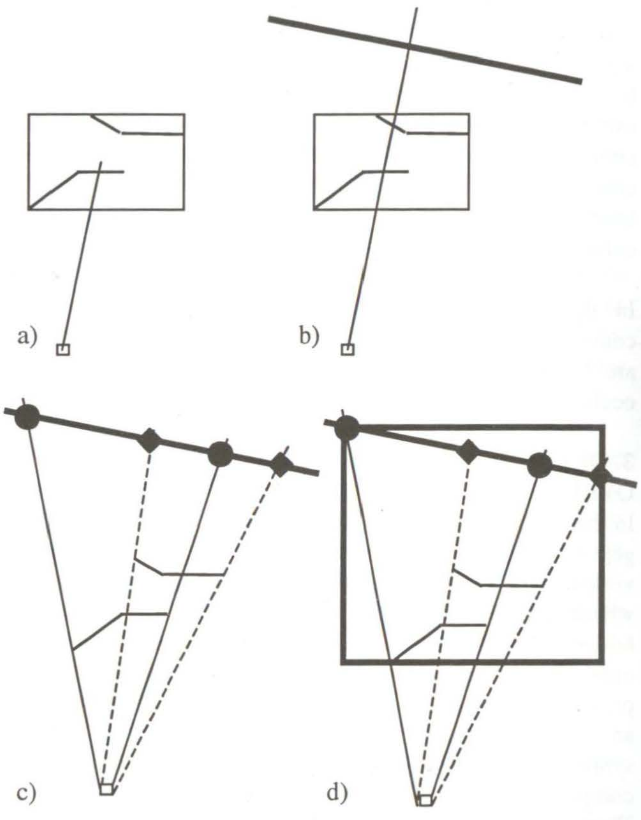


Fig. 5: The computing of the bounding box.

Once the root of the octree and the separating planes of the first hoop are computed, we proceed to subdivide the octree using those planes. The nodes of the octree that are in the positive half-space of all the planes are inside the shadow and are classified as *black* nodes; the nodes that are outside the shadow are classified as *white*. The algorithm processes incrementally each separating plane (*update_octree* function) using a recursive top-down octree traversal.

For the initial plane, an octree node (the octree's root initially) is classified respect to the separating plane, if it is traversed by the plane then it is classified as *grey*. *Grey* nodes are subdivided and its eight children classified recursively. When the maximum depth of the octree is reached, the *grey* nodes are conservatively classified as *white*. If a node is inside the positive half-space of the plane, it is classify as *black*, and *white* in the opposite case.

The additional separating planes are processed in a similar way: a new octree traversal is needed and to define the node's type (*update_node* function) it is mandatory to take into account the current classification of the node against the plane and its previous type, according to the criteria shown in table 1. Whatever the previous classification of a node was, if the current classification is TRAVERSED, then the node is finally classified as GREY.

Images (a), (b), (c) in figure 6 show in 2D the first steps of the algorithm for the first separating planes. Images (d) and (e) show the resulting octree for two planes labeled 1 and 2. Image (f) show the final result once all the separating planes have been processed.

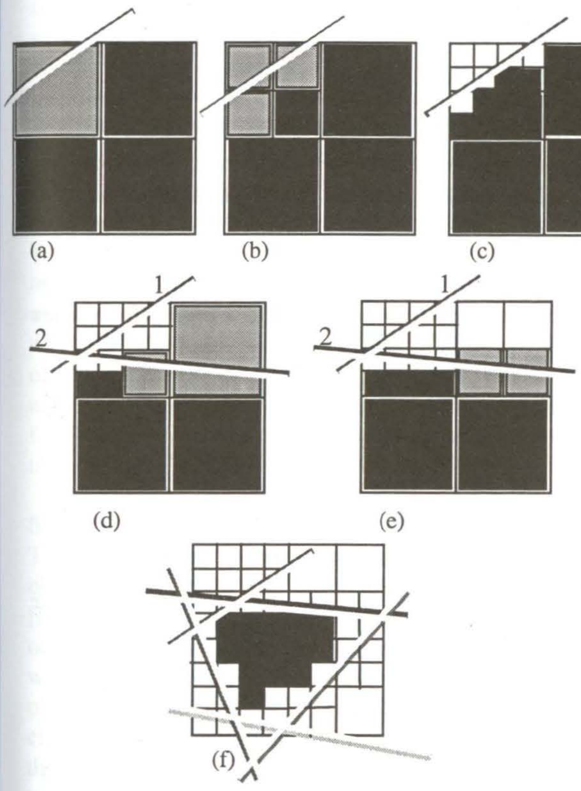


Fig. 6: The computing of the octree of the shadow of the first hoop. The lined side of each plane represents its positive half-space.

Every time a previously WHITE or BLACK node is subdivided its final classification becomes GREY, and its eight children get an indetermined classification, represented by the label INDET_0. The algorithm detects if a node does not have a classification and it assigns the current classification of the node against the plane to the node. Note that at the end of the algorithm, all the nodes that were classified as INDET_0 have a classification of WHITE, BLACK or GREY.

```
Procedure    update_octree(O:octree;    p:plane;
maxdepth:integer)
var root:node;
begin
      root:=get_root(O); // the boundig box
       //1 is the depth of the root
       //maxdepth   is   the   maximum   depth   of
       //subdivisions
       update_node(O, root, p, 1, maxdepth);
end;
```

```
procedure update_node(O:octree; n:node; p:plane;
depth, maxdepth:integer)
  var cl: integer; k:node;
  begin
   if node_has_children(n) then
   begin
    for each son k of n do
      update_node(O, k, p, depth+1, maxdepth);
   end
   else
   begin
     cl:=classify_node(n,p);
     case (cl) of
       OUTSIDE:
        if      get_prev_class(n)='INDET_0'       or
        get_prev_class(n)='BLACK' then
            set_final_class(n,'WHITE');
       INSIDE:
         if get_prev_class(n)='INDET_0' then
            set_final_class(n,'BLACK')
       TRAVERSED:
         if depth<maxdepth then
         begin
          set_final_class(n, 'GREY');
          subdivide_node(O, n);
          for each son k of n do
          begin
            set_final_class(k,'INDET_0');
            update_node(O,k,p,depth+1,maxdepth);
          end;
         end
         else
          set_final_class(n, 'OUTSIDE');
      end;
    end;
  end;
```

Algorithm 2: Construction of the octree for the first hoop.

The algorithm is easy and efficient due to the low number of hoop's edges. The shadow octree of this first hoop can also be used to select the hoops that must be fused with it, i.e., the hoops that intersect this shadow octree.

4.3 Update the Octree with the Second Hoop.

The result of the previous algorithm is the octree of the shadow of the initial hoop, the nodes of which have been classified as BLACK, WHITE or GREY. The proposal of the updating function is to fusion this octree with the shadow cast by another hoop to obtain the shadow octree of the union of the shadow. The proposed algorithm updates the nodes of the first octree in order to get, in the end, black nodes where the region of the space is inside either the shadow of the first hoop or the second hoop, white nodes where the region of the space is outside both shadows and grey nodes at the shadow boundary of the union.

We proceed as follows: the separating planes of the second hoop from the view-cell are computed and, for each of them, the octree is traversed (*update_octree2*) in a pre-order traversal. The nodes are classified (*update_node2*) taking into account the previous and the current classification of the node. We compute the final classification according to table 2.

Previous classif.	Current classif.	Final Classif.
WHITE	OUTSIDE	WHITE
WHITE	INSIDE	INSIDE
WHITE	TRAVERSED	GREY
BLACK	OUTSIDE	BLACK
BLACK	INSIDE	BLACK
BLACK	TRAVERSED	BLACK
INSIDE	OUTSIDE	WHITE
INSIDE	TRAVERSED	GREY

Table 2: Classification of the nodes for the second hoop.

In the case of a GREY node being found, nothing is done with it, just its children are updated recursively.

In the case of a previously classified BLACK node, its final classification is BLACK whatever the current classification against the plane is.

As the shadow of the second hoop is computed incrementally using the separating planes, special care must be taken when the nodes previously classified as WHITE are classified as INSIDE against the current plane. These nodes are provisionally classified as INSIDE. To consider these nodes inside the shadow, they must be classified INSIDE against all the separating planes.

In the case of a node being traversed by the current separating plane, then, whatever the previous classification was WHITE or INSIDE, if the maximum depth has not been reached, the node is subdivided, labeled as GREY and its eight children classified recursively. If the maximum depth has been reached, the node is conservatively classified as WHITE. The children of a subdivided previous terminal node get an indetermined classification represented by the label INDET_1. The algorithm detects if a node does not have a previous classification and the current classification is assigned directly to it. Note that in the end of the algorithm, all the nodes that were classified as INDET_1 have a classification of WHITE, BLACK or GREY.

The final BLACK nodes will be nodes classified as BLACK for the hoop1 and nodes classify INSIDE of the hoop2. This is taken into account in the compression process. If there are more than two hoops to fuse, then for each new hoop, we have to use diferent INSIDE codes and consider the previously classifide INSIDE nodes as BLACK.

Fig. 7 shows a 2D example of the updating of the octree. Black nodes are the ones that were classified BLACK with the previous algorithm, grey nodes are inside the shadow of the second hoop.

```
Procedure update_octree2(O:octree; p:plane;
maxdepth:integer)
var root:node;
begin
     root:=get_root(O);
     update_node2(O, root, p, 1, maxdepth);
end;

procedure update_node2 (O:octree; n:node;
p:plane; depth, maxdepth:integer)
var cl, prev_class: integer; k:node;
begin
 if node_has_children(n) then
  for each son k of n do
    update_node2(O, k, p, depth+1, maxdepth);
 else
 begin
    cl:=classify_node(n,p); //current class.
    prev_class:=get_prev_class(n);   //previous
//  classification.
   case (cl) of
    OUTSIDE:
     If (prev_class = 'WHITE') or (prev_class
     = 'INSIDE') or (prev_class = 'INDET_1')
     then
        set_final_class(n,'WHITE');
    INSIDE:
      if prev_class='WHITE' then
         set_final_class(n,'INSIDE');
      if prev_class='INDET_1' then
         set_final_class(n,'BLACK');
    TRAVERSED:
      if depth<maxdepth then
      begin
        if  (prev_class   =   'INSIDE')   or
     (prev_class = 'WHITE') then
       set_final_class(n, 'GREY');
       subdivide_node(O, n);
       for each son k of n do
       begin
        set_final_class(k,'INDET_1');
        update_node2(O,k,p,depth+1,maxdepth);
       end;
      end
      else
       set_final_class(n, 'OUTSIDE');
    end;
  end;
end;
```

Algorithm 3: Construction of the octree for the second hoop.

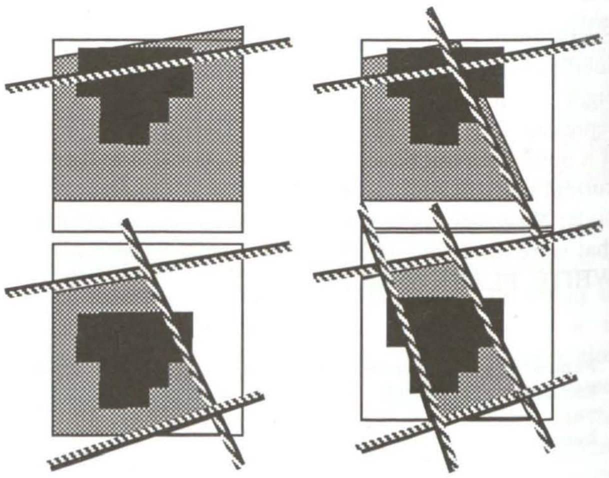


Fig. 7: The updating of the octree with hoop 2. Black and grey areas represent the union of the octrees of the shadows.

4.4 Compression

As can be easily noticed, the separating planes always produce the subdivision of the nodes that they cross if the octree maximum depth is not reached. So both inside and outside nodes can be over-subdivided. In order to make hoop extraction more efficient and save memory, the octree is compressed using a bottom-up traversal. For each node, we check the classification of its eight children. If all of them are WHITE, then the node turns its classifications from GREY to WHITE and its eight children destroyed. The same if its children are BLACK. In case one or more of its children is classified as GREY, then nothing is done and the algorithm continues.

We have found experimentally that it is not efficient to apply the octree compression after each separating plane updating. Compression becomes necessary when the umbra is computed for more than four hoops (the number of hoop edges is usually less than 10); and it is mandatory the compression previously to compute the final hoop (because the hoop construction algorithm takes advantage of having large black nodes).

5. EVALUATION AND RESULTS

The implemented algorithm works with any kind of 3D object whose hoop has been previously synthesized by [Brunet01]. The only auxiliary data structure used is an octree to represent the shadow of the hoop (note that it was also needed an octree to compute a hoop for a given object). There are only two geometrical tests: the classification of a point and a box respect to a plane. Note that it is not necessary to compute the intersections, only to detect them. So the algorithm is robust and simple.

The synthetic data used to test the algorithm was the St. Paul's Cathedral[1] model from where the octree and its hoop were extracted using [Brunet01] algorithm. The scene was formed by two cathedrals located one behind the other as shown in fig.8 and fig. 9. All the algorithms were programed in C++ and tested on a Pentium II 400Mhz.

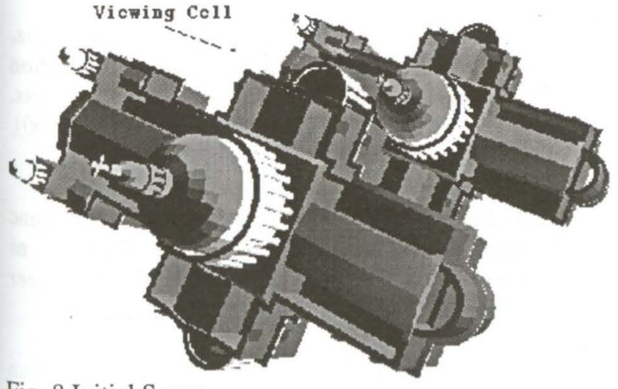


Fig. 8 Initial Scene.

[1] Publicly available at *http://www.3dcafe.com*

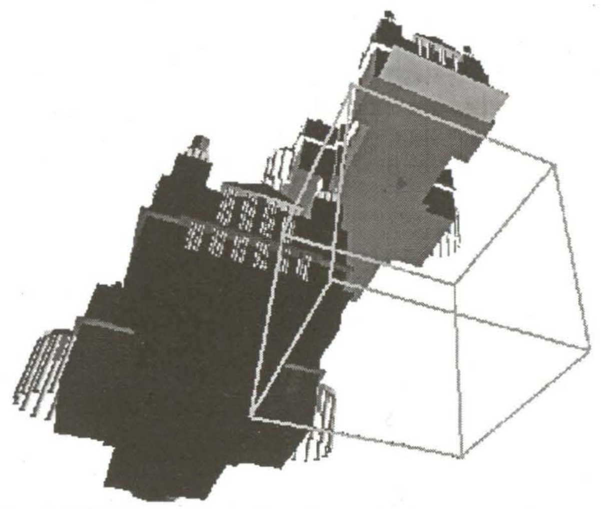


Fig. 9 Initial scene from the view-cell (in grey)

Table 3 shows the number of nodes for the shadow octree of the union of two hoops (one for each cathedral) given a view-cell (see figure 8). The statistics have been computed for different depths of the octree and before and after the compression process. InH0 and InH1 are the nodes classified as BLACK respect to the first and second hoops respectively: they are considered as simple BLACK nodes after the compression. InH0 includes the nodes inside the first hoop and in the intersection of the two shadows. Note that the degree of compression is notable and that less terminal nodes are obtained. Nevertheless, we have found that for the efficiency of the global algorithm, the compression must be done after a number of fusions (usually 10). Also note that more compression is obtained for WHITE nodes, that corresponds with the algorithm's design.

Table 4 shows the number of nodes of the shadow octree for a hoop obtained as a result of the fusion of the previous ones. From the shadow octree of the fusion, a hoop was obtained with the algorithm in [Brunet01] and its shadow octree has been computed using the same octree bounding box and view-cell. Note that the compression of the resulting octree does not affect the number of internal nodes. This is due to the algorithm design and that the hoop is seen convex from the cell. Also note that the final number of nodes is similar to the number of nodes of the shadow octree obtained by the fusion algorithm. If the size of the nodes is taken into account, then it can be concluded that the resulting hoop covers neraly the same that the fusion shadows. This fact is shown in fig. 9. Each hoop has been projected from each vertex of the view-cell onto a plane located behind the cathedrals. The intersection of all the projections for one hoop is, in fact, the projected shadow of the hoop. These projections show the projected shadows of the original hoops labeled as 1 and 2 and the projected shadow of the final hoop in magenta. As can be seen the resulting hoop of the union (labeled as R) covers as much as the other two. This means that the shadow of the resulting hoop is as big as the shadows of the original hoops together, so the two hoops can be substituted by the resulting one with a good conservative visibility occlusion.

The bigger the resolution of the octree, the more accurate is the resulting hoop. Note how the shadow R covers the corners of the other shadows as the resolution of the octree grows.

The processing times for the fusion algorithm are reasonable (interactive rates) taken into account that the hoop fusion will be a pre-process for an algorithm of cell to cell visibility or for the computation of the potential visible set [Saona-Vazquez99].

6. CONCLUSIONS AND FUTURE WORK

We have presented an efficient method to compute the hoop fusion of a set of hoops. It allows to substitute a set of objects the shadows of which overlap from a view-cell by a single occluder for visibility computations from a view-cell. The resulted occluder has better occlusion performances than the individual hoops of the initial occluders.

The algorithm is based on a simple and robust algorithm that compute incrementally the shadow octree of a set of hoops. From this octree it is possible to obtain the fusion hoop by applying a previously published construction process [Brunet01].

The experimental results demonstrate the goodness of the proposed technique. As seen in the examples the hoop of the shadow union covers as much as the hoops of the shadow of the two objects together, solving then the visibility problem.

Our future work is related to integrate the fusion to our cell-to-cell visibility test to compute pre-process visibility [Saona-Vazquez99]. Moreover, we would like to analyze the effect of considering the terminal *grey* nodes as *black* nodes instead of *white* ones. A priory this moves the algorithm from conservative (underestimate) to approximate (overestimate) respect to the occluder visibility computations [Andujar00b], but it could be useful to fuse occluders that are far enough from the view-cell so it is not possible to see between them. Note that in this case, it could also be possible to reduce the octree resolution so improve the algorithm efficiency. Also note that the initial hoops are always inner to the object so the degree of approximation could be relatively low.

7. ACKNOWLEDGEMENTS

The authors would like to thank C. Saona-Vazquez and J. Jou for their previous research and implementations on hoops. This work has been partially supported by the Spanish Ministry of Science and Technology (grant TIC2001-2226-C02-01).

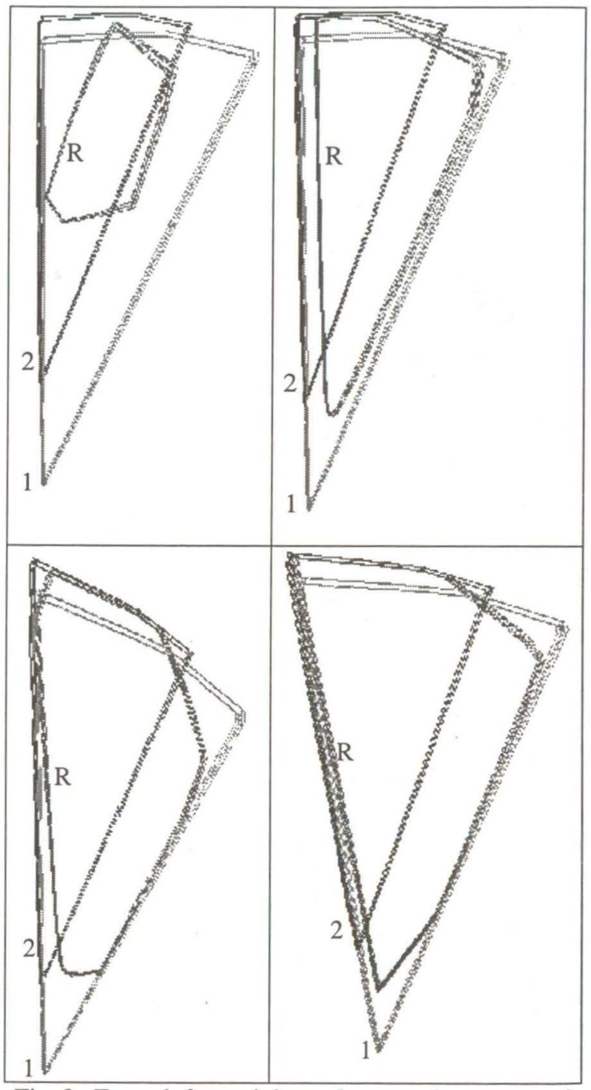


Fig 9. From left to right and top to bottom: projected shadows for maximum depth 4, 5, 6 and 7

8. REFERENCES

[Andujar00] Carlos Andujar, Carlos Saona-Vázquez and Isabel Navazo. LOD visibility culling and occluder synthesis. *Computer-Aided Design,* 32(13):773-783, November 2000.

[Andujar00b] Carlos Andujar, Carlos Saona-Vázquez, Isabel Navazo and Pere Brunet. Integrating occlusion culling with levels of detail through hardly-visible sets. *Computer Graphics Forum (Eurographics'00),* 19(3):499-506, August 2000.

[Brunet01] Pere Brunet, Isabel Navazo, Jarek Rossignac and Carlos Saona-Vázquez. Hoops.3D Curves as Conservative Occluders for Cell-Visibility. *Computer Graphics Forum 20(3)* pp 431-442, September 2001.

[Cohen-Or98] Daniel Cohen-Or, Gadi Fibich, Dan Halperin and Eyal Zadicario, Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum,*17(3):243-253, 1998

[Durand00] Frédo Durand, george Drettakis, Joëlle Thollor and Claude Puech. Conservative visibility preprocessing using extended projections. In *SIGGRAPH 2000 Proceedings,* pages 239-248, July 2000.

[Koltun00] Vladlen Koltun, Yiorgos Chrysanthou and Daniel Cohen-Or. Virtual occluders: An efficient intermediate PVS representation. In *Eurographics Workshop on Rendering,* pages 59-70, June 2000.

[Law99] Fei-Ah Law and Tiow-Seng Tan. Preprocessing occlusion for real-time selective refinement. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics,* pages 47-53, 1999.

[Saona-Vazquez99] Carlos Saona-Vázquez, Isabel Navazo and Pere Brunet. The visibility octree. A data structure for 3D navigation. *Computer & Graphics,* 23(5):635-643,1999.

[Schaufler00] Gernot Schaufler, Julie Dorsey, Xavier Decoret and François X. Sillion. Conservative volumentric visibility with occluder fusion. In *SIGGRAPH 2000 Proceedings,* pages 229-238, July 2000.

[Wonka00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. *Rendering Techniques 2000 (Proceedings of the Eurographics Workshop on Rendering 2000).* pages 71–82. June 2000.

[Zhang98] Hansong Zhang, *Effective occlusion culling for the interactive display of arbitrary models.* PhD thesis, University of North Carolina at Chapel Hill, 1998.

Depth	Octree of the union									
	Before Comp.						After Comp.			
	InH0	InH1	Tot	Out	Grey	Tot	W	B	G	Tot
4	29	108	137	277	59	473	137	102	34	273
5	257	700	957	1263	317	2537	577	600	168	1345
6	1200	3206	4406	5731	1448	11585	2455	2642	728	5825
7	5094	13223	18317	27408	6532	52257	10447	10743	3027	24217

Table 3: Nodes of the octree of the union of 2 hoops

Depth	Resulting octree							
	Before Comp				After Comp.			
	B	W	G	Tot	B	W	G	Tot
4	64	378	63	505	64	154	31	249
5	565	1907	353	2825	565	731	185	1481
6	2027	11225	1893	15145	2027	3042	724	5793
7	7877	21930	4258	34065	7877	8889	2395	19161

Table 4: Nodes of the octree of the resulting hoop