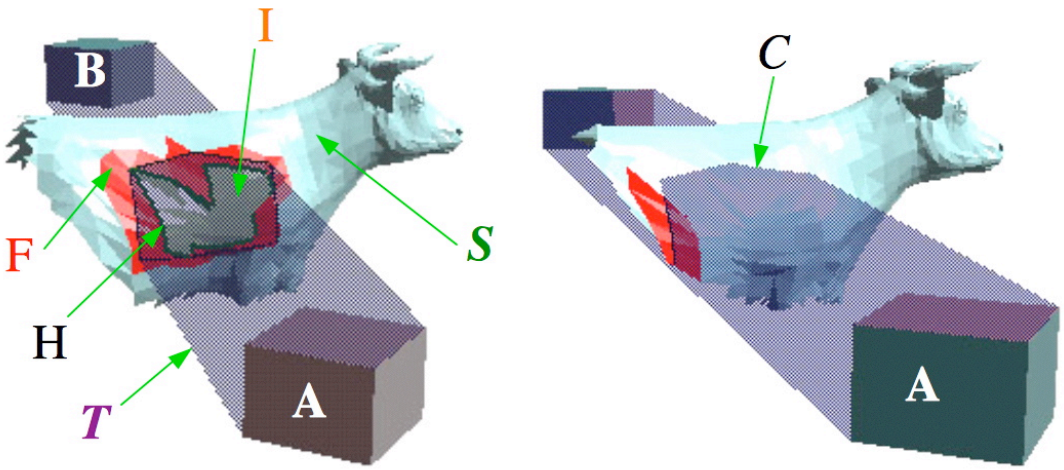

Figure 1: *On the left, we can establish that the half-cow surface, **S**, is occluding cube* **A** *from cube* **B**, *because its intersection with the transparent tubular surface, **T**, contains a closed loop, C, that bounds a subset **D** of **S** that has no hole and an even number of intersections with an arbitrary ray joining the two cubes. The set* F *of triangles stabbed by C is painted red. They are separated from the rest of **D** by edges from the set* H *(fat dark lines), which are computed during the walk around C to control the process of invading the set* I *of triangles completely contained in **D**. On the right, we cannot guarantee occlusion, because C reaches a bounding edge of **S**, and hence does not form a closed loop on **S**. Only those triangles of* F *that have been traversed before reaching the border of **S** are painted red.*

ShieldTester: Cell-to-cell visibility test for surface occluders

Isabel Navazo[(a)], Jarek Rossignac[(b)], Joan Jou[(a)], Rahim Shariff[(b)]

[(a)] Computer Graphics Group, Polytechnic University of Catalonia, Barcelona, Spain
[(b)] GVU Center, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, USA

Abstract

We present a novel Cell-To-Cell Visibility (C2CV) algorithm, which given two polyhedra, **A** *and* **B**, *and a connected and oriented manifold triangle mesh, **S**, offers a simple, fast, and conservative test for detecting when* **A** *and* **B** *are occluded from each other by **S**. Previously disclosed C2CV algorithms either relied on costly occlusion fusion or were restricted to convex (or "apparently convex") occluders, which makes them inappropriate for scenes where potential occluders are arbitrary triangulated surfaces, such as the body of a car or a portion of a terrain. The simplicity of our C2CV algorithm, named ShieldTester, stems from a new Occlusion Theorem, introduced here which permits to establish occlusion by computing the intersection of **S** with a single ray from a vertex of* **A** *to a vertex of* **B**. *ShieldTester may be used to establish that pairs of cells in a subdivision of space are hidden from each other by a relatively large surface occluder, so that when the viewer is in one cell, the objects in the other cell need not be displayed.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Computational geometry and Object Modeling: Occlusion Culling, Visibility Test, Triangle Meshes.

1-Introduction

The rapidly growing complexity of the scenes and models used in interactive 3D environments imposes a considerable burden on the graphics subsystem. Mesh simplification, texture mapped imposters, and panoramas have provided considerable relief to the vertex processing bottleneck of contemporary rendering pipelines, when rendering complex scenes. Still, in many common situations, the majority of the objects of the scene are not visible in any given view. If undetected, they increase significantly the vertex-processing and pixel–processing cost for each frame. They may also increase transmission delays when viewing remote 3D environments in Internet applications. The majority of small objects may be easily culled because their bounding spheres do not intersect the viewing frustum. Still, in scenes with high depth complexity, such as cities, ships, aircrafts, factories, and interiors, graphic performance may be significantly increased if one can avoid the cost of rendering a large fraction of the objects that intersect the viewing frustum, but are occluded by other objects [2, 21, 26]. Consequently, occlusion testing, also called visibility detection, has received a large amount of attention in recent years.

An object is said to be visible if any part of it is visible. Otherwise, it is said to be occluded. Occlusion tests may be exact (reporting precisely the set of occluded objects), conservative (reporting a subset of the occluded objects), or approximate (possibly reporting as occluded some visible objects). Occlusion may be computed from a given viewpoint ("from-point-visibility") at rendering time or pre-computed for an entire cell ("from-cell-visibility"). We say that an object **B** is occluded from cell **A** when it is occluded from all view-points in **A**. Instead of testing the visibility, from a

given cell **A**, of a large set of objects, one may test whether a cell **B** is occluded from **A**, in which case, all objects in **B** are also occluded. Note that, if **B** is occluded from **A**, then **A** is also occluded from **B**. "Cell-to-cell-visibility", abbreviated C2CV, may be pre-computed between all pairs of cells in a regular or irregular partition of space. Initially, exact C2CV algorithms were restricted to test occlusion by a single occluder, which had either to be convex or to appear convex from all view-points in the cells. Recently developed occlusion fusion establishes exact C2CV for arbitrary combinations of polygonal occluders. To do so, it performs, for each occluder, a series of Boolean operations in five dimensions.

In contrast, the Occlusion Theorem and the associated C2CV algorithm implementation, named *ShieldTester,* that we present here cast a single ray, R_{AB}, from a vertex of **A** to a vertex of **B**. If R_{AB} does not intersect the candidate occluding triangle mesh, ***S***, we correctly report that **A** is not occluded by ***S*** from **B**. If R_{AB} intersects one or more triangles of ***S***, we start a WALK from a subset of these "stabbed" triangles. The WALK traces a connected component of the intersection of ***S*** with the boundary of the convex hull of the union of **A** and **B** (Fig. 1). All geometric computations used to identify the triangles stabbed by R_{AB} and to perform the WALK are reduced to testing the signs of 3x3 determinants. If the WALK closes a loop, we count how many stabbed triangles it has crossed and how many stabbed triangles are inside the portion of ***S*** bounded by that loop. If the total is odd, we guarantee occlusion.

Note however that *ShieldTester* is conservative, in that when it reports occlusion or visibility, it is exact, but it may also report that occlusion has not been established, when in fact ***S*** hides **A** from **B**. Furthermore, ShieldTester can only test for occlusion by single connected triangle mesh occluder and performs no occlusion fusion. Thus, it is only useful when the occluding surface is large compared to the two cells. Such occluder candidates include: terrain models; large panels in the body of a car, aircraft, photocopier; the hull of a ship; and the curtain of a theater. Finally, for simplicity, we assume that **A**, ***S***, and **B** are mutually disjoint and that **A** and **B** are convex, although this assumption may be easily lifted by replacing ***S*** by ***S***–(**A**⊕**B**) and by replacing non-convex cells by their convex hulls or bounding boxes.

The paper is organized as follows. We first review prior art. Then, we derive and prove the Occlusion Theorem, upon which our approach is based. Then, we provide an overview of the ShieldTester algorithm for triangular meshes, followed by the implementation details. The naming and typesetting conventions used throughout the paper are summarized in the Appendix.

2-Prior Art

To provide a context for our work, we include a brief overview of related techniques. More detailed classification and review of occlusion techniques may be found elsewhere [17].

Several authors have proposed to perform dynamic from-point-visibility tests by checking, for each frame, whether a simple bound enclosing an object or a group of objects lies in the region hidden by one or several occluders. This test can be performed in three dimensions [5, 11] or in image space [2, 9, 10, 24, 26]. For example, an occlusion wall may be constructed using the z-buffer information produced by rendering one or several occluders located near the current viewpoint. Objects whose enclosing bounds are hidden by that wall need not be rendered. To simplify and speed up the occlusion tests, geometrically simple convex occluder candidates are identified (manually or algorithmically) during a preprocessing phase [24, 26] and occluders that are sufficiently large or sufficiently close to the viewer are used [5, 26]. Although such dynamic occlusion approaches are sometimes very effective, they rarely find an optimal compromise between the computing cost per frame and the rate of successful detection of hidden objects [2, 10]. Thus, in this paper, we focus on techniques for pre-computing cell-to-cell occlusion, given a set of static occluders. Note that a pre-computed visibility approach may be combined with dynamic occlusion, especially in scenes with large moving occluders [22].

In the presence of a single convex occluder, **O**, an object **B** is invisible from all viewpoints in a polyhedral cell **A**, if it is hidden by **O** from all the vertices of **A** [6, 16, 20]. To exploit this powerful property, most of the pre-processing or dynamic occlusion culling algorithms consider only simple and convex occluders, for which the occlusion of **B** from the vertices of **A** may be easily tested. Therefore, a variety of approaches have been proposed to generate simple and convex occluders that may be safely used instead of the original shapes in performing occlusion tests. These occluder impostors support a conservative test, because they lie inside the original solids or behind them, but inside the region they occlude [1, 4, 12, 14, 21]. Unfortunately, most objects in a typical scene are either not simple or not convex, especially when the scene is composed of surface models that, either by design or as a result of algorithmic choices or errors, do not form proper boundaries of closed sets. Therefore, in this paper, we focus on algorithms for directly detecting occlusion by a surface, rather than by a solid.

Instead of testing the visibility of all objects from a cell, **A**, given a set of chosen occluder candidates, it may be advantageous to pre-compute the visibility between cells. Cell-to-cell visibility (C2CV) may be pre-computed for the cells of a partition of space and used to quickly extract which parts of the scene are visible, as the viewer moves from one cell to an adjacent one [6, 20]. Although elegant approaches that compute exact cell-to-cell visibility have been proposed [8, 23], they are computationally expensive [17] for complex and general 3D scenes and have been often restricted to 2.5D scenes of urban or terrain models [3, 13].

Given two cells, **A** and **B**, and an occluder ***S***, **A** is hidden from **B** by ***S*** if all rays joining a point of **A** to a point of **B** intersect ***S***. We use the notation **A**|***S***|**B** to indicate such situations. Note that **A**|***S***|**B** is equivalent to **B**|***S***|**A**. **A**|***S***|**B** may be tested efficiently when ***S*** is *convex* and disjoint from **A** and **B**. In this case, all the rays that join vertices of **A** to the

vertices of **B** have to intersect ***S*** [6, 20]. A simple proof is based on the fact that, when a point **a** ∈ **A** is hidden by a convex occluder ***S*** from points $\mathbf{b}_1 \in \mathbf{B}$ and $\mathbf{b}_2 \in \mathbf{B}$, then it is also hidden from all points of the line segment $(\mathbf{b}_1,\mathbf{b}_2)$. Our contribution is to relax this convexity restriction on ***S*** and, more importantly, to allow ***S*** to be a curved surface with boundary and possibly non-zero genus. Thus ***S*** does not need to enclose a solid and may have one or more holes and handles.

The Hoops approach [4] generalizes the concept of occluder convexity to that of "*apparent convexity*" and justifies the use of the efficient ray-shooting approach mentioned above whenever a portion $\boldsymbol{D} \subset \boldsymbol{S}$ is found that appears convex when viewed from any point in **A**. The following simple test was proposed in [4] for detecting when ***D*** is apparently convex. When walking around the polygonal border of an apparently convex mesh ***D*** in one of the two possible directions, we make turns at each vertex that all appear to be left turns from all points of **A**. In other words, the plane through a vertex **v** of the border of ***D*** and through the two neighbors on that border separates space into points that see a left turn at **v** and points that see a right turn at **v**. Hence, ***D*** will appear convex from **A** if **A** lies in the intersection of linear half-spaces, each bounded by a plane associated with a different vertex of the boundary of ***D***. The cell **A** lies in the intersection of half-spaces if each one of its vertices lies inside each half-space. The main challenge of the Hoops approach is to produce good candidates for ***D***, for which the authors have used a special octree decomposition of the model bounded by ***S***. The ShieldTester approach proposed here provides a deterministic approach for constructing good candidates for ***D*** directly from the intersection of ***S*** with the boundary of the convex hull of **A** ∪ **B**. It also does not require testing the vertices of **A** or **B** against the half-space defined above.

Often, groups of small objects may together form an occluder that hides one cell from another, while none of the individual objects does. Hence, the results of occlusion tests on individual objects may not be easily combined to test for occlusion by the group. To address these situations, several occluder-fusion algorithms have been developed. Some of these approaches are restricted to 2.5D scenes [12, 25] and require occluders to have a well defined interior of sufficient thickness, providing enough room to construct simple convex blockers that are contained in the interior of these occluders. For example, the approach of [21] will not work for occluders that are surfaces with boundaries or even for very thin curved solid walls. The image space approach proposed in [7] is the first one to cope with concave occluders. Its cost is significantly higher than for convex occluders and it is restricted to closed surfaces. An exact approach to polygonal occluder fusion has been recently proposed by Nirenstein et al. [17] using Boolean operations in the five-dimensional Plücker space to carve out the contribution of each occluder. Note that this exact approach requires one such 5D Boolean operation per triangle of a curved surface. In comparison, the conservative, although not exact, ShieldTester approach, presented here, reduces this cost to a small number of cross-products and dot-products for a relatively small subset of these triangles and to a simple traversal for another subset of the triangles.

3-Occlusion Theorem

We assume that the candidate occluding surface, ***S***, is a connected and oriented two-manifold surface with or without boundary. (Later we will further restrict it to be an edge-connected triangle mesh.) Thus, ***S*** may have holes and handles. For simplicity, we assume that the two cells, **A** and **B**, are mutually disjoint and convex. Furthermore, we assume that ***S*** does not intersect them.

The purpose of this section is to formulate a practical and sufficient condition that guarantees occlusion. Because ***S*** is not convex it is not sufficient to test whether ***S*** intersects all rays from a vertex of **A** to a vertex of **B**. Yet, surprisingly, the Occlusion Theorem discussed in this paper require only computing the intersection between ***S*** and a single ray R_{AB} from **A** to **B**.

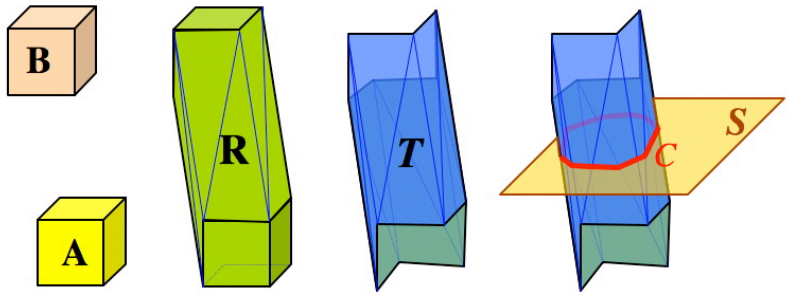

Figure 2: *Consider the two cubic cells* **A** *and* **B** *(left). The convex hull* **R** *of their union (second from left) shares some faces with the cells. Its other faces form the tube* ***T*** *shown as a transparent surface (third from left). The intersection C (red curve right) of* ***T*** *with a surface* ***S*** *may bound an occluding portion* ***D*** *of* ***S****.*

Let **R** denote the union of all rays from **A** to **B**. Because **A** and **B** are convex, **R** is the convex hull of **A** ∪ **B**. As a proof: consider that the convex-hull of a 3-D point set is the union of all tetrahedra with vertices in it. Thus tetrahedra with 4 vertices in **A** lie in **R**. A tetrahedron with vertices, **a**, **b**, and **c**, in **A** and vertex **d** in **B** forms a pencil of all rays from **d** to points in triangle *(***a**,**b**,**c***)*, which lies in **A**, since **A** is convex. Thus the pencil of rays lies in **R**. A tetrahedron with

vertices, **a** and **b** in **A** and vertices, **c** and **d**, in **B** is the union of rays from a point in **A**, on edge *(***a**,**b***)*, to a point in **B**, on edge *(***c**,**d***)***,** is contained in **R**. We have proven that the convex hull of **A**∪**B** is a subset of **R**. A ray from a point in **A** to a point in **B** is obviously in the convex hull of **A**∪**B** and therefore **R**⊇**A**∪**B**.

Let the tube ***T*** be the portion of the boundary of **R** that is disjoint from **A** and from **B** and let J be the intersection ***S***∩***T***, which for simplicity, we assume to be one-dimensional. If we find a maximally connected curve component *C* in ***S***∩***T*** (see Fig. 2) that satisfies the conditions listed in the following Occlusion Theorem, we conclude occlusion (**A**|***S***|**B**). Otherwise, we report that we cannot guarantee occlusion.

Occlusion Theorem: The surface ***S*** occludes cell **A** from cell **B**, if and only if:

1) There exists a connected component, *C*, of ***S***∩***T*** that is bounding a portion ***D*** of ***S*** and has no boundary in the interior of **R**. (In other words, the portion, ∂***D***∩**R**, of the boundary of ***D*** that lies in **R** is exactly *C*.)

2) Given an arbitrary ray, R_{AB}, from a point, **a**, in **A** to a point, **b**, in **B**, R_{AB} has an odd number of none tangential intersections with the subset, ***D***∩*C*, of ***S***.

Proof. Assume that R_{AB} joins a point **a** of **A** to a point **b** of **B** and has an odd number of none tangential intersections with ***D***∩*C*. We will show that the two conditions imply that any other ray, (**a**',**b**'), from **A** to **B** will also intersect ***D*** (Fig. 3). Consider a continuous linear motion of point **a** towards **a**' (Fig. 3 bottom). During that motion, the ray R_{AB} sweeps the triangle (**a**,**a**',**b**). The intersection of that triangle with ***S*** contains a curve which, by hypothesis, has an odd number of transversal (non-tangential) intersection points with the edge (**a**,**b**). Given that ***D*** has no boundary in **R**, the curve does not have a bounding vertex in the triangle. It cannot intersect the edge*)* **(a**,**a**'**),** because edge (**a**,**a**') is in **A** and ***S*** is disjoint from **A**. Therefore, the curve must leave the triangle through the edge (**a**',**b***)* and must thus have an odd number of "crossing" intersections with that edge. (We are not counting non-crossing tangential contacts where the curve osculates the edge without traversing it.) Reversing the role of **A** and **B** and using the same argument on triangle (**a**',**b**,**b**'), we prove that it also has an odd number of crossing intersections with (**a**',**b**'*)*. We have proven that any ray from **A** to **B** has an odd number of intersections with ***D***. Because the number of intersections cannot be negative, it must be at least 1.

Note that ***D*** may be a valid occluder even when it has handles and a boundary (other than *C*), provided that the boundary lies outside of **R**. Fig. 4 shows different topological situations between ***S*** and ***T***, the Occlusion Theorem handles all of them correctly.

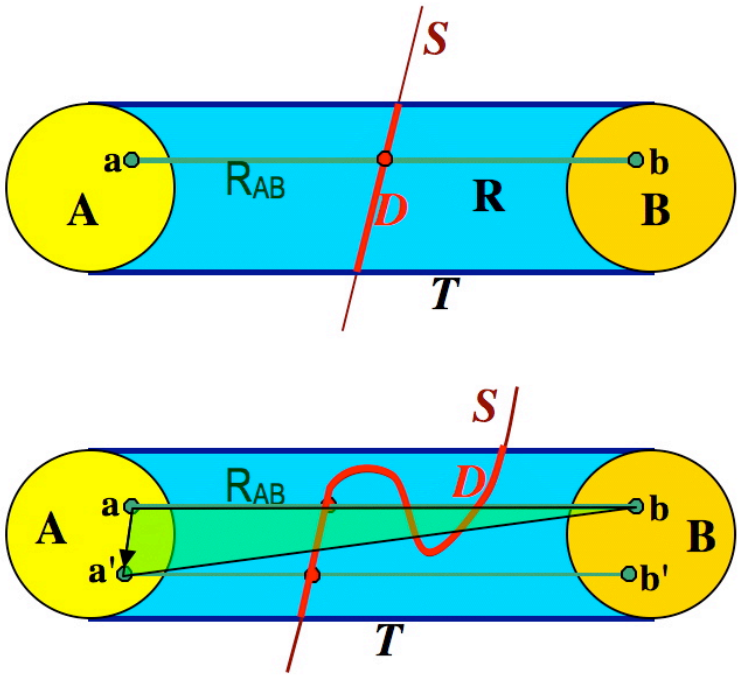

Figure 3: *In the top figure, the ray* R_{AB}*, from a point* **a** *of* **A** *to a point* **b** *of* **B** *intersects* ***D*** *only once. Clearly* **A**|***D***|**B**. *In the bottom figure,* R_{AB} *intersects* ***D*** *three times. Note that the parity of this number of intersections is preserved for all rays obtained by continuously moving* **a** *to* **a**' *or* **b** *to* **b**' *as long as, during that motion, the ray does not cross the boundary of* ***D***.

The naïve implementation of the Occlusion Theorem could be organized as follows.

```
Compute S∩T
FOR EACH component C of S∩T Do {
    IF C is a closed loop THEN DO {
        FOR EACH subset D of S that contains C in its boundary DO {
            IF D has no hole in R THEN IF RAB intersects D∩C an odd number of times THEN RETURN ("occluded") }}}
RETURN ("undecided")
```

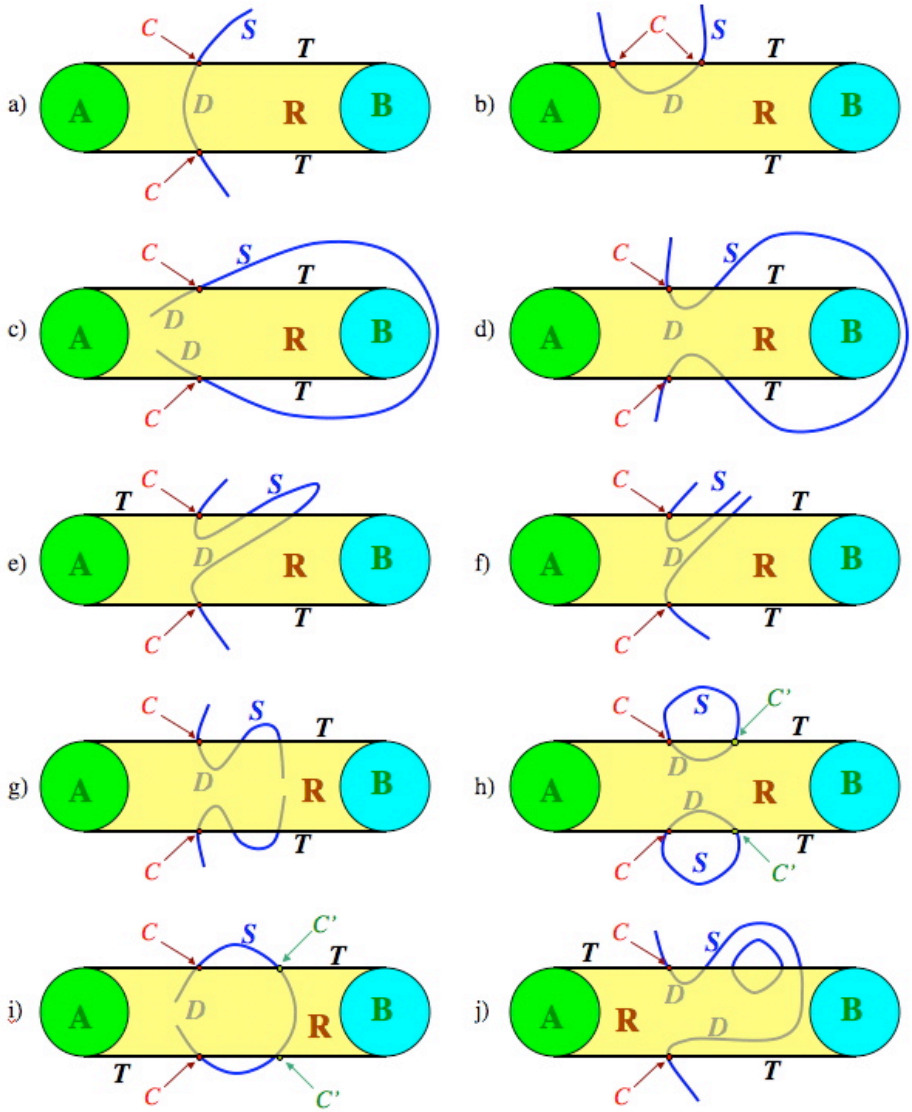

Figure 4: *We illustrate the variety of possible configurations in which C forms a closed loop around a component **D** of **S**. Occlusion by **D** may be guaranteed in case (a), (e) and (j) because **D** has no hole and* R_{AB} *has an odd number of intersection points with **D**. Note that **D** needs not be completely contained in **R**, as illustrated in (e), and may have a handle (j). In the other cases, occlusion by **D** may not be reported because parity is even (b, d, h) or because **D** has a hole (c, f, g, i). Note that in the (i) case, occlusion could be established if we were to use the component **S–D** instead of **D** or when another component, C', of* $S \cap T$ *is processed. Note that the Occlusion Theorem would allow us to correctly conclude occlusion for (f) while the more conservative implementation of ShieldTester, described below, will not.*

4-ShieldTester customization for triangle meshes

Although our Occlusion Theorem is valid for arbitrary surfaces with boundary, the simple and efficient ShieldTester implementation, presented in this paper, is restricted to the practical and important subset of triangulated and oriented manifold surfaces with or without boundary. ShieldTester uses several improvements over a naïve implementation of the Occlusion Theorem. These improvements, justified in more details in the next section, were developed to eliminate some unnecessary steps and to avoid the cost of computing the intersection curve, $S \cap T$.

The high-level algorithm for ShieldTester is organized as follows:

```
Obtain an edge e_AB of T from a vertex of A to a vertex of B ;
IF e_AB ∩ S = ∅ THEN RETURN (“visible”) ;
FOR EACH triangle x of S that is stabbed by e_AB DO {
    Start a WALK that traces a connected component C of S ∩ T
    IF the WALK did not run into a boundary of S THEN {
            INVADE the portion of S to the left of the triangles crossed by the WALK along C
            IF the invaded portion has no hole THEN
                    IF the number of triangles stabbed by e_AB and visited by WALK or INVADE is odd THEN
                            RETURN (“occluded”) } }
RETURN (“undecided”)
```

The specific improvements used in ShieldTester over a naïve implementation of the Occlusion Theorem are:

1) The first condition stated in the Occlusion Theorem ($\partial D \cap \mathbf{R}=C$), is replaced by a weaker (more conservative, but cheaper to compute) condition ($\partial D=C$). In other words, there exists a subset, ***D***, of **S** that has no holes and has for only boundary the connected component, *C,* of $S \cap T$.
2) Instead of using a different ray R_{AB} from **A** to **B** for each ***D*** candidate, we use for all the tests a single edge e_{AB} of ***T*** that joins a vertex of **A** to a vertex of **B**. We compute and mark, once and for all, the triangles of ***S*** that are stabbed by e_{AB}.
3) To establish whether e_{AB} stabs a particular triangle of ***S***, we evaluate the sign of five 3x3 determinants.
4) The WALK procedure starts at a triangle of ***S*** stabbed by e_{AB} and walks along the intersection $S \cap T$. It does not actually compute $S \cap T$. Instead, it keeps track of the triangle t or ***T*** and s of ***S*** on which it lies at any given moment during the traversal of *C*. It uses the signs of a few 3x3 determinants to decide whether it first exits s or t, and through which edge.
5) To avoid having to split the triangles of ***S*** that are crossed by *C*, we add them to ***D*** and test occlusion by this extended set ***D***'.
6) We separate the triangles of ***D***' into two sets (see Fig. 1): the set F of (so called "crossed") triangles that have been traversed by a WALK process and the set I of the remaining (so called "interior") triangles, which are then visited by an INVADE process. Note that the triangles of I need not lie in **R**.
7) Note that I may have several connected components. To ensure that all are visited by INVADE, a superset H of the bounding edges of I is created during the WALK. These edges are selected from the edges of the triangles of F that are "to the left" of *C*. The concept of being "to the left" of *C* is defined more precisely later. Informally, *C* splits each triangle f of F into two or more parts. The parts "to the left" of *C* intersect **R**. The other parts do not. Some edges in the set H that are not bounding I are marked as "cold".
8) The parity of the number of intersections of e_{AB} with the closed subset $D \cap C$ of ***S*** is reduced to toggling a parity bit during the WALK (each time a stabbed triangle in F is encountered) and during the topological invasion (procedure INVADE) of the set I of the interior triangles of ***D*** (each time a stabbed triangle of I is encountered).

A more detailed formulation of the complete ShieldTester algorithm is presented in following pseudo-code, explained below. In particular, it outlines the logic of the WALK process and the call to INVADE. It maintains several flags and counters for keeping track of the parity of the number of stabbed triangles and to indicate whether a hole has been encountered.

```
e_AB := edge of T from a vertex of A to a vertex of B;
X := set of triangles of S stabbed by e_AB ;
IF (X is empty) THEN RETURN ("visible");
FOREACH triangle x in X DO
    IF NOTx.crossed THEN {
        k++;                            # Id of next candidate
        parityIsOdd := FALSE;
        walkedIntoHole := FALSE;
        e:= e_AB;  t:=x;                # starting vertex of C
        REPEAT {
            (e,t) := Step(e,t);     # next vertex along C
            IF (e is an edge of S) THEN UpdateWalk;
            } UNTIL ((e,t)==( e_AB,x) OR walkedIntoHole);
        IF NOT walkedIntoHole THEN
            FOREACH edge e of H DO
                IF (NOT e.cold)  AND (NOT t_N.invaded == k)
                    THEN IF NOT Invade(e,k)
                        THEN IF (parityIsOdd)
                            THEN RETURN ("occluded"); }
RETURN("uncertain");
```

Each vertex of $S \cap T$ is identified by an edge e of ***S*** or ***T*** and by the triangle t of ***T*** or ***S*** (respectively) stabbed by e. Thus the starting vertex of *C* is identified by the pair (e_{AB},x). The call "step(e,t)" returns another pair, also called (e,t), which defines the next vertex along $S \cap T$.

When, during the WALK, we cross an edge of ***S*** ("e is an edge of ***S***"), we perform a series of updates ("UpdateWalk"). In particular, we mark as "crossed" the triangle of ***S*** that has been entered by crossing that edge, so as to prevent starting another walk from it ("IF NOT x.crossed THEN"). We also report when we have encountered a border of ***S*** (by setting the variable "walkedIntoHole" to TRUE). Finally, we add to the set H the edges that are possibly bounding I. We also mark

some edges as "cold", so as to avoid adding them later to H. We may also mark as "cold" some edge that have already been added to H but are not part of the boundary of I. They will be skipped during "Invade". The WALK finishes when we return to the starting vertex (e_{AB},x) of *C* or when a hole was reached.

If we have made a complete loop ("IF NOT walkedIntoHole"), we call "Invade(e,k)" for each edge e of H that has not been marked as "cold" during the WALK and bounds on its left a triangle t_N that has not yet been invaded. Each call to "Invade(e,k)" performs a topological traversal of the edge-connected subset of the triangles of I that is incident upon edge e. The parameter "k" is used to mark all of the triangles visited by invade for a particular *C*, so as to avoid visiting them twice for the same candidate *C*.

"Invade" toggles "ParityIsOdd" each time it visits a stabbed triangle of X. It returns TRUE, if a border edge was reached.

In the remainder of this paper, we explain how e_{AB}, X, and H are computed. We also discuss the details of how we identify "cold" edges. Finally, provide the details of UpdateWalk and Invade(e,k).

IMPLEMENTATION DETAILS

Finding an edge e_{AB} of *T*

First, we need to obtain an edge e_{AB} of T that joins a vertex of **A** to a vertex of **B**. We outline below 3 options.

If **A** and **B** are axis-aligned cubes of a regular space partition, then e_{AB} is implicitly defined by the relative position of **A** and **B** on the 3D grid. For example, if **B** is in the positive octant with respect to **A**, then the segment joining the vertices (0,0,1) of **A** and of **B**, in their local coordinates, may be used as e_{AB}.

For more general polyhedral cells, we have developed two different approaches. The first one computes the convex hull of **A**∪**B** and selects an edge of it that joins a vertex of **A** with a vertex of **B**.

The second implementation avoids building the convex hull explicitly. Instead, it finds an extreme vertex **a** of **A**∪**B**. Then, we place a supporting plane at **a** and rotate it around **a**, until it encounters another vertex **b** of **A**∪**B**. If **a** is on **A** and **b** on **B** (or vise versa), we set e_{AB} =(**a**,**b**). Otherwise, assume without loss of generality that **a** and **b** are on **A**. We roll the plane around **A** while keeping all the vertices of **A** on one side of it, until we encounter the first vertex of **B**. At this point, we have a triangle of the convex hull of **A**∪**B** that has two vertices on **A** and one on **B**. Thus, we have two suitable candidates for e_{AB}. Furthermore, we can rotate the plane around any one of these two edges until we hit another vertex of **A** or **B**. We have used this process to generate the triangles and edges of ***T***, on the fly, as they are needed by the WALK algorithm.

Identifying the set X of triangles of *S* stabbed by e_{AB}

Let s(**a**,**b**,**c**,**d**) be TRUE when (**ab**×**ac**)•**ad** > 0, where **ab** stands for **b**–**a**. Note that evaluating s(**a**,**b**,**c**,**d**) amounts to computing the sign of a 3x3 determinant, whose 9 coefficients are each the difference between the coordinates of two vertices.

Edge (**a**,**b**) intersects triangle (**c**,**d**,**e**), if and only if s(**c**,**d**,**e**,**a**) = s(**b**,**c**,**d**,**e**) = s(**b**,**d**,**e**,**a**) = s(**c**,**b**,**e**,**a**) = s(**c**,**d**,**b**,**a**). One possible configuration of such an intersection is shown in Fig. 5. If s(**c**,**d**,**e**,**a**) = s(**b**,**c**,**d**,**e**), then **a** and **b** are on opposite sides of the plane through **c**, **d**, and **e**. The other three expressions test that **b** lies inside the frustum of rays from **a** that stab the triangle.

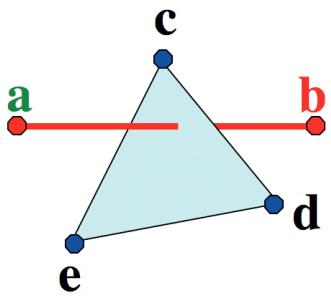

Figure 5: *Edge* (**b**,**a**) *stabs triangle* (**c**,**d**,**e**) *if and only if* s(**c**,**d**,**e**,**a**) = s(**b**,**c**,**d**,**e**) = s(**b**,**d**,**e**,**a**) = s(**c**,**b**,**e**,**a**) = s(**c**,**d**,**b**,**a**).

Thus, given the vertices **a** and **b** of e_{AB} and the vertices **c**, **d**, and **e** of a triangle t of ***S***, we call STAB(**a**,**b**,**c**,**d**,**e**), which evaluates the signs of (a sufficient subset of) the five 3x3 determinants listed above to decide whether t is "stabbed" or not by e_{AB}. In our implementation, we do this for all of the triangles of ***S***.

A fractions of the triangles may be rejected early, after computing only the first two expressions, s(**c**,**d**,**e**,**a**) and s(**b**,**c**,**d**,**e**), if they yield different results. However, if numerous pair of cells need to be tested for mutual occlusion by ***S***, it may be advantageous to pre-compute a BSP tree or other hierarchical data structure that would permit a quick rejection of large groups of triangles, because the region of space that contains them is not stabbed by e_{AB}.

For simplicity, the algorithm described here assume that no edge of ***T*** intersects an edge or vertex of ***S***. Proper processing of such "degenerate" intersection as special cases would not only severely complicate the code, but would

also require a thorough treatment of numeric precision problems. It has been the subject of a large body of research and falls beyond the scope of this work. Instead, we have opted for a "lazy" approach, which consists of testing for possibly degenerate cases using a numeric tolerance. When a degenerate situation is found, we declare conservatively that we are unable to establish whether **A|*S*|B** or not. To reduce the probability of running into a singular situation, we perturb the vertices of ***T*** by a small random displacement that will enlarge ***T***, and thus preserve the validity of the results. To perform the perturbation, the coordinates of the vertices of ***T*** are normalized to an enclosing axis-aligned bound and quantized to a fixed number of bits, chosen so as to ensure that no edge collapses. We then apply a sufficiently small random perturbation to each coordinate that brings the associated vertex outside of ***T***. We use long integers to represent the perturbed vertex coordinates in a suitable unit and evaluate the signs of all 3x3 determinants exactly. In rare cases, where the perturbed situation still exhibits a geometric singularity (i.e., some 3x3 determinant is zero), we could perform a second random perturbation and try again. We do not, and simply declare that **A|*S*|B** could not be established.

Making a "Step" from one vertex of *S*∩*T* to the next along *C*

The WALK procedure is a state machine that performs a sequence of steps. Each step appends to the current end, **p**, of the curve *C* a new edge joining **p** to a new end-vertex, **q**. This new edge lies inside a triangle s of ***S*** and inside a triangle t of ***T***. Let (**a**,**b**,**c**) be the vertices of triangle s and (**d**,**e**,**f**) be the vertices of triangle t. Assume, without loss of generality, that, as shown in Fig. 6, **p** is the intersection of edge (**b**,**c**) of s with triangle (**d**,**e**,**f**) of t.

Assuming that the two triangles are in general position (no degenerate situations), we have only 5 possibilities for **q**:

1) it is the intersection of triangle (**d**,**e**,**f**) with edge (**a**,**b**), as shown Fig. 6, left;
2) it is the intersection of triangle (**d**,**e**,**f**) with edge (**a**,**c**);
3) it is the intersection of triangle (**a**,**b**,**c**) with edge (**e**,**f**), as shown Fig. 6, right;
4) it is the intersection of triangle (**a**,**b**,**c**) with edge (**f**,**d**),
5) it is the intersection of triangle (**a**,**b**,**c**) with edge (**d**,**e**),

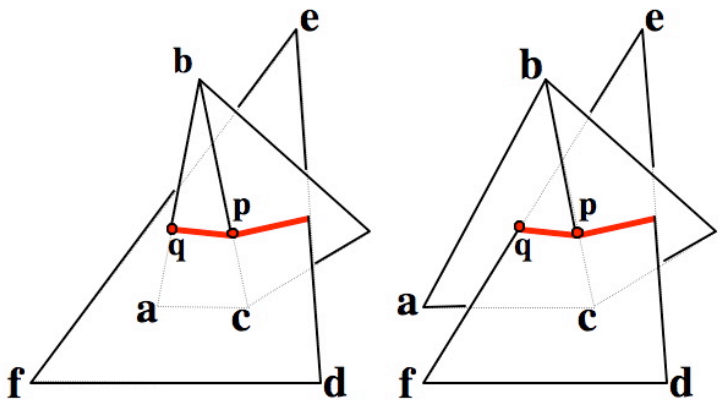

Figure 6: *When* STAB(**a**,**b**,**d**,**e**,**f**) *the next segment* (**pq**) *of C ends at an edge of triangle* s *(left). When* STAB(**e**,**f**,**a**,**b**,**c**) *is true it ends at an edge of triangle* t *(right).*

We distinguish between these situations, we use the "STAB" test and the degeneracy handling approach described in the previous subsection. In particular, if STAB(**a**,**b**,**d**,**e**,**f**), then **q** is on edge (**a**,**b**), see Fig. 6 left. Otherwise, if STAB(**a**,**c**,**d**,**e**,**f**), **q** is on edge (**a**,**c**). Otherwise, **q** is on an edge of triangle (**e**,**d**,**f**). Note that because **p** lies inside (**e**,**d**,**f**), only one edge of (**e**,**d**,**f**) can stab (**a**,**b**,**c**). Thus, if STAB(**d**,**e**,**a**,**b**,**c**), then **q** lies on edge (**d**,**e**). Otherwise, if STAB(**e**,**f**,**a**,**b**,**c**) then **q** lies on edge (**e**,**f**). Otherwise, **q** lies on edge (**f**,**d**).

Note that we never need to actually compute the geometric coordinates of the new vertex **q** of *C*. The "Step" process described above takes as input an edge e of ***S*** (or ***T***) and a triangle t of ***T*** (or ***S***) that define the current end-vertex, **p**, of *C* where e intersects t. From these, it extracts the six points used above and returns the edge and triangle pair that intersect at **q**.

Characterization of the "left of *C*" and of the cold edges

This subsection establishes how to decide which portion of ***S*** lies to, what we call, the "left of *C*". Furthermore, it justifies that it is sufficient to analyze that portion (i.e. to visit the triangles it contains through an Invade process, in order to test that the invaded region has no hole, and to count the number of stabbed triangles encountered during this invasion). In particular, it explains which edges are included in H and which are marked as "cold".

Let **p** denote the end point of the portion of *C* reconstructed so far. Let **o** denote the outward normal to ***T*** at **p** and let **n** denote the normal to ***S*** at **p**. These two normal vectors are defined by the triangle of ***T*** and the triangle of ***S*** that have been entered most recently by the *Walk*. The tangent direction of the next edge from **p** along *C* is **u**=**n**×**o** (see Fig. 7). The left of *C*, with respect to ***S*** is defined by the direction **left**=**n**×**u**. Thus, **left**=**n**×(**n**×**o**)=(**n**•**o**)**n**−(**n**•**n**)**o**, and therefore **left**•**o**=(**n**•**o**)^2−(**n**•**n**)(**o**•**o**)=(**n**•**o**)^2−1, which is always negative. Because ***T*** is convex, the plane through **p** with normal **o** is a supporting plane that cannot be crossed by *C*, because *C* is constrained to remain on ***T***. Consequently, any deviation

from the direction **u** has to be in the opposite direction to **o**, and hence to the left of *C*. In conclusion, *C* only makes left turns on ***S***, with respect to the local coordinate frame established using the tangent to *C* and the outward normal to ***S***.

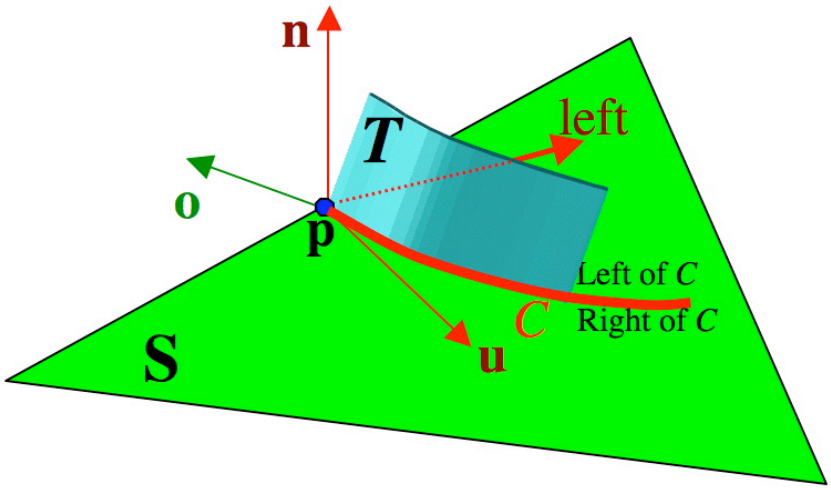

Figure 7: *The curve C is oriented in the direction* ***u=n o****, defined as the cross-product of the outward normal,* ***o****, to the tube* ***T*** *with the outward normal* ***n*** *to* ***S****, both taken at point* ***p****. C turns always left on* ***S****, away from the outward normal* ***o*** *to the tube* ***T****.*

The curve *C* can either split ***S*** into two components (i.e. see Fig. 4-a) or be a "topological cut" [15] that does not split ***S***, but reduces its genus (see Fig. 4-h). Bellow we define an interior, ***D***, of *C* which will be valid for testing occlusion in both these situations.

When *C* splits ***S*** into two components, $\boldsymbol{D_1}$ and $\boldsymbol{D_2}$, each one could possibly satisfy the Occlusion Theorem (see Figs. 4-a and 4-i), and therefore one may consider testing both components. However, we need only to test the component that approaches *C* from the interior of **R**, which is to the left of *C* on ***S***. We call that component the *"interior"* of *C*. Note that the interior of *C* needs not be entirely contained in **R** (Fig. 4-e), but only an infinitely small offset of *C* on ***S*** towards its interior (i.e., towards the left) must be in **R**. Assume now that the interior of *C* is not a valid occluder (Fig. 4-i), but that the other component (called the *"exterior"* component) of ***S*** is. Although that exterior surface component is not processed here, it must contain another component *C'* of $\boldsymbol{S} - \boldsymbol{T}$, whose interior satisfies the Occlusion Theorem (Fig. 4-i). Thus, it will be discovered and processed later when ShieldTester tests *C'*, unless of course occlusion by another portion of ***S*** is established earlier.

The Occlusion Theorem and its proof remain valid, even when *C* is a topological cut on ***S*** that does not separate ***S*** into two components, but instead bounds the same component ***D*** on both sides. For example, *C* could be a non-contractible curve on a torus, (see Fig. 4-h). Such not-occluding situations are rejected by the Occlusion Theorem. Thus, our test works correctly, even when such topological cuts are not distinguished from curves that split ***S***.

In conclusion, it is sufficient to analyze the interior ***D*** of *C* defined as the region of ***S*** that lies to the "left of *C*" on ***S***.

Next, we explain how to compute, while visiting the triangles of F, the list H of edges that contain the border of I. Assume that during the walk, we have crossed an edge, e, of ***S***, going from the previously crossed triangle, t_P, of S to the next crossed triangle, t_N, of S. Use the notation shown in Fig. 8. Because C does not turn right, the two edges e_R and e'_R must be cold. (They are either crossed by C or lie to its right.) However, e_L may be a valid edge in the boundary of the set I of interior triangles to ***D***. Thus, we mark e, e_R, and e'_R as cold and if e_L is not cold, we add it to the list H.

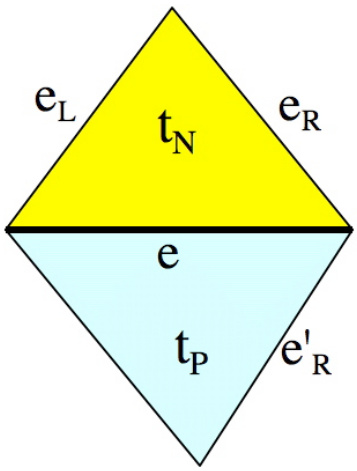

Figure 8: *Edge e was reached from triangle* t_P.

Note that, as shown in Fig. 9, there is only one situation where a triangle f of F contributes an edge to H that is not marked as "cold".

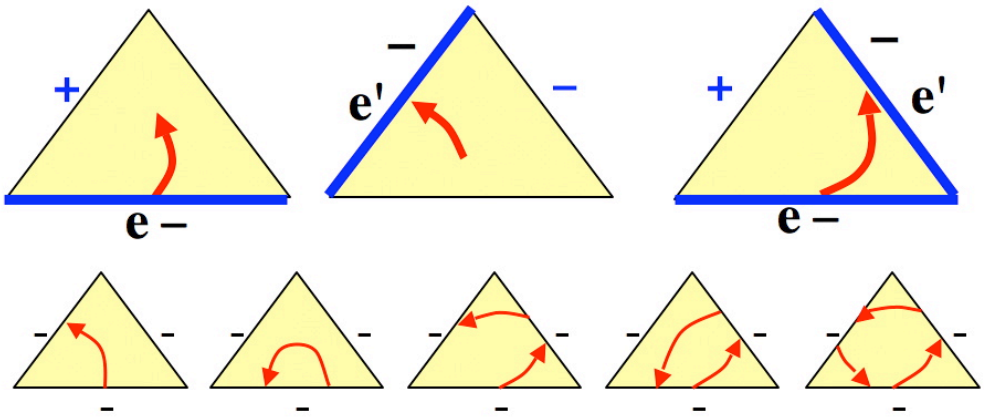

Figure 9: *During a Walk, when entering a triangle through edge e (top left) we mark e as cold (minus sign) and if the edge on the left to e has not been already marked as cold, we put it in the list H of candidate hot edges (plus sign). When leaving a triangle through edge e' (top center), we mark e' and its successor in clockwise order as cold (minus signs). The only situation where an edge is put in the list H and not marked as cold is shown top right. All other types of single or multiple crossings (shown in the bottom row) mark all edges as cold.*

Updating things during the WALK

Given an edge e, and using the notation of Fig. 8, the call to UpdateWalk performs the following operations:

```
IF (tN does not exist) THEN walkedIntoHole := TRUE
ELSE {   e.cold := TRUE;
         eR.cold := TRUE;
         e'R.cold := TRUE;
         IF (NOT eL.cold) THEN add eL to H;
         IF ((NOT tN.crossed) AND (tN.stabbed)) THEN ParityIsOdd := TRUE;
         tN.crossed := TRUE; }
```

Invading the set I of interior triangles

When **A** and **B** are axis aligned cubic cells of a regular partition of space, the tube ***T*** (see Fig. 2) is simple and the test whether an edge on the boundary of ***D*** lies inside **R** may be reduced to a 2D test against a simple convex polygon, which is the intersection of **R** with a plane that bisects the line segments joining the centers of **A** and **B**. Thus, our first implementation of the Occlusion Theorem tested that ***D*** has no border inside **R**.

However, when **A** and **B** are complex, the complexity of ***T*** may significantly increase the cost of testing whether a border edge of ***S*** lies inside **R**. Therefore, we have developed, in ShieldTester, an alternative implementation that is based on a purely topological traversal of I and checks that I has not borders, except for the hot edges.

In other words, if during Invade we reach a border edge of ***S***, we assume that it could be inside **R**, and hence simply discard the corresponding portion ***D*** of ***S***. Although this "lazy" approach may reject some good occluders (see Fig. 4-f), it guarantees not to accept bad ones (see Fig. 4-b, c, d, g, h).

Using the notation of Fig. 8, the pseudo-code for Invade(e,k) is:

```
Procedure Invade(e,k) {
        IF (tN does not exist) THEN RETURN(TRUE); # found hole
        IF (NOT ((tN.invaded==k) OR tN.crossed)) THEN {
                IF (tN.stabbed) THEN parityIsOdd := NOT parityIsOdd;
                tN.invaded := k;
                RETURN (Invade(eL,k) OR Invade(eR,k)); }
        ELSE RETURN (FALSE)}
```

Note that the set I of interior triangles may have more than one connected component. All components will be visited, because each one is incident upon at least one edge of H that is not cold.

6-Performance considerations

Because **R** is convex, a triangle of ***S***, cannot be entered by all the WALKs more than 3 times. Therefore, the worst case asymptotic cost of the walks is linear in the number of triangles of **S**. So is the cost of computing the set X of triangles stabbed by e_{AB}.

However, the topological Invade process may visit a triangle of S at most once per connected component of $S \cap T$. Thus, the worst-case asymptotic complexity of the calls to Invade, taken together, is quadratic in the number of triangles in ***S***. In practice, of course, the number of components in $S \cap T$ is small.

We have produced two independent implementations of ShieldTester and have tested them on several surface models for a variety of cell configurations. Some examples are shown in Figs. 1 and 10.

Although our implementations have not been optimized in any way, they are efficient (see statistics in Fig. 10).

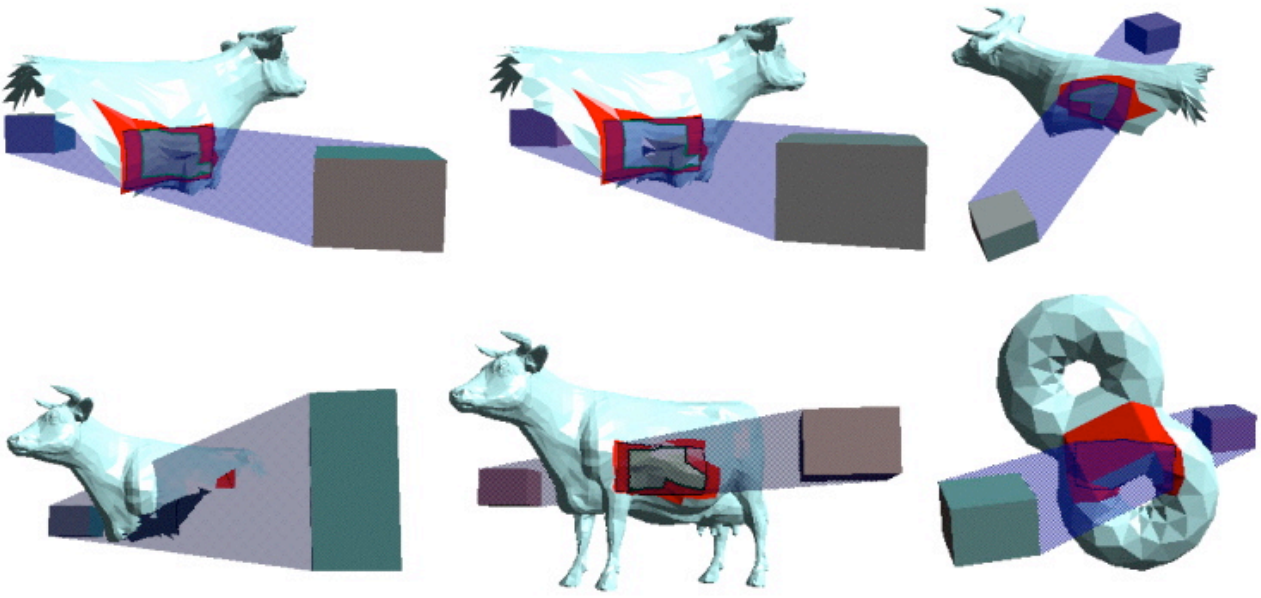

Figure 10: *The first four models have all about 3000 triangles. The fifth model has 5000 triangles. The numbers of triangles crossed by all the WALKs for each one of the first five models are respectively 52, 52, 25, 15, and 51. The total numbers of invaded triangles for these models are respectively 39, 18, 9, 0, and 37. Occlusion was successfully established in cases 1, 3, and 5. Each occlusion test took respectively 17, 14, 15, 14, and 31 millisecond.*

7-Conclusions and future work

We have presented a new sufficient condition for the occlusion of two arbitrary sets **A** and **B** by an arbitrary two-manifold surface, ***S***, which may have handles and holes. Based on this condition and on several innovative solutions, we have developed an efficient topological approach for testing whether an oriented, manifold, triangle mesh is an occluder for a pair of convex cells. It should be stated that our solution does not require that we compute any geometric intersection explicitly. Instead, we simply test a series of mixed products involving vectors between the vertices of **A**, **B** and ***S***. The approach is fast and simple to implement. The proposed ShieldTester solution may be used to pre-compute the visibility between pairs of cells in a regular decomposition of the scene in the presence of a set of static obstacles. A non-optimized implementation requires less than 50 milliseconds to perform a cell-to-cell visibility tests for moderate complexity occluders, such as those shown Fig.10.

We were not able to compare ShieldTester to exact visibility algorithms that perform occluder fusion, such as the one of [17]. Such a study would help to establish whether it is always appropriate to use ShieldTester for large surface occluders, or whether an exact approach is preferable.

References

1. C. Andujar, C. Saona-Vázquez and I. Navazo. “LOD visibility culling and occluder synthesis”, Computer Aided Design, **32** (13): 773-783, 2000.
2. D. Bartz, M. Meissner, and T. Hüttner, “ OpenGL-assisted occlusion culling for large polygonal model”, *Computer & Graphics* **23**(5): 667-679, 1999.
3. J. Bittner and P. Wonka, “Visibility preprocessing for urban scenes using line space subdivision”, *19th Pacific Conference on Computer Graphics and Applications Proc.*: 276-283, 2001.
4. Pere Brunet, Isabel Navazo, Jarek Rossignac and Carlos Saona-Vázquez, “Hoops: 3D curves as conservative occluders for cell-visibility”, *Computer Graphics Forum (Eurographics’01 Proc.)*, **19** (3): 499-506, 2000.
5. S. Coorg and S. Teller. “Real-time occlusion culling for models with large occluders”, *Proceedings of the Symposium on Interactive 3D Graphics*: 83-90, 1997.
6. D. Cohen-Or, G. Fibich, D. Halperin, and *E*. Zadicario, “Conservative visibility and strong occlusion for view space partitioning of densely occluded scenes”, *Computer Graphics Forum (Eurographics’98 Proc.),* **17**(3):243-253, 1998.
7. F. Durand, G. Dretakkis, J. Thollot and C. Puech, “Conservative visibility preprocessing using extended projections”, *Proceedings of SIGGRAPH’00*: 239-248, 2000.

8. F. Durand, G. Dretakkis and C. Puech, "The visibility complex", *ACM Transactions on Graphics* **21**(2): 176-206, 2002.
9. N. Greene, and N.Kass, "Hierarchical z-buffer visibility", *Proceedings of SIGGRAPH'93*: 231-240, 1993.
10. H. Hey, R. F. Tobler and W. Purgathofer, "Real time occlusion culling with a lazy occlusion grid", *Proceedings of the 12th Eurographics Workshop on Rendering*, 2001.
11. T. Hudson, D. Manocha, J. Cohen, M. Ling, K. Of. and H. Zhang, "Accelerated occlusion culling using shadow frusta", *Proceedings of the 13th Symposium on Computational Geometry*: 1-10, 1997.
12. V. Koltun, Y. Chrysanthou and D. Cohen-Or., "Virtual Occluders□An efficient intermediate PVS representation", *Proceedings of the 11th* Eurographics Workshop on Rendering: 59-70, 2000.
13. V. Koltun, Y. Chrysanthou and D. Cohen-Or, "Hardware-accelerated from-region visibility using a dual ray space", *Proceedings of the 12th Eurographics Workshop on Rendering*, pp. 205-216, 2001.
14 F.-A. Law and T.-S. Tan, "Preprocessing occlusion for real-time selective refinement", *Proceedings of the ACM Symposium on Interactive 3D Graphics*: 47-54, 1999.
15 W. Massey, "Algebraic Topology: An Introduction", *Graduate Texts in Mathematics* **56**, Springer-Verlag, 1977.
16. B. Nadler, G. Fibich, S. Lev-Yehudi and D. Cohen-Or, "A qualitative and quantitative visibility analysis in urban scenes", *Computer & Graphics* **23** (5): 655-666, 1999.
17. S. Nirenstein, E. Blake and J. Gain,□Exact From-region visibility culling", *Proceedings of the 13th Eurographics Workshop on Rendering*, 2002.
18 J. Rossignac and M. O'Connor, "SGC: A Dimension-independent model for pointsets with internal structures and incomplete boundaries", *in Geometric Modeling for Product Engineering*, Eds. M. Wosny, J. Turner, K. Preiss, North-Holland: 145-180, (Proceedings of the IFIP Workshop on CAD/CAM), 1989.
19. J. Rossignac, A. Safonova and A. Szymczak, "3D compression made simple: Edgebreaker on a Corner Table", *Proceedings of the Shape Modeling International Conference*: 278-283, 2001.
20. C. Saona-Vázquez, I. Navazo and Pere Brunet, "The visibility octree: A data structure for 3d navigation", *Computer & Graphics*, **23** (5): 635-644, 1999.
21. G. Schaufler, J. Dorsey, X. Decoret and F.X. Sillion, "Conservative volumetric visibility with occluder fusion", *Proceedings of SIGGRAPH 2000*: 229-238, 2000.
22. O. Sudarsky and C. Gotsman, "Dynamic Scene Occlusion Culling", *IEEE Transactions on Visualization and Computer Graphics*, **5** (1): 13-29, 1999.
23. S.J. Teller, "Computing the antipenumbra of an area light source", *ACM Computer Graphics (Proceedings of SIGGRAPH 92)* **26**: 139-148, 1992.
24 P. Wonka and D. Schmalsteig, "Occluder shadows for fast walkthroughs of urban environments", *Computer Graphics Forum,* **18**(3): 51-60, 1999.
25. P. Wonka, M. Wimmer and D. Schmalstieg, "Visibility preprocessing with occluder fusion for urban walkthroughs", *Proceedings of the 11th Eurographics Workshop on Rendering*,: 71-82, 2000.
26. H. Zhang, D. Manocha, T. Hudson and K. Hoff, "Visibility culling using hierarchical occlusion maps", *Proceedings of SIGGRAPH'97*: 77-88, 1997.

Appendix

We use the following typesetting convention.

- **SOLID**: Bold capitals refer to solids.
- ***SURFACE***: Bold italic capitals refer to surfaces.
- *CURVE*: Italic capitals refer to curves.
- COLLECTIONS: Capital letters that are not bold and not italic refer to collections of edges or triangles.
- elements: Lower case not bold letters refer to topological elements such as vertices, triangles, or edges.
- **points**: Lower case bold letters refer to points (which may represent the locations of vertices of the mesh) and vectors.
- (**a**,**b**) will denote the line segment joining the points **a** and **b**.
- (**a**,**b**,**c**) will denote the triangle with vertices **a**, **b**, and **c.**

We use the following symbols in the paper.

- ***S*** denotes an arbitrary manifold surface or a triangle mesh considered to be a potential occluder
- **A** and **B** denote the two convex cells for which we test occlusion by ***S***.
- **R** denotes the convex hull of **A**□**B**, which is the union of all rays from **A** to **B**.
- ***T*** denotes the portion of the tubular boundary of **R** that is not bounding **A** and **B**.
- *J* denotes the curve where ***T*** and ***S*** intersect. Note that *J* may have several connected components.
- *C* denotes a maximally connected component of the curve *J* of intersection between ***T*** and ***S*.**
- ***D*** denotes a portion of ***S*** that is bounded by *C*.
- R_{AB} denotes a ray from a point in **A** to a point in **B**.
- e_{AB} denotes a particular instance of R_{AB}, that is an edge of ***T*** joining a vertex of **A** to a vertex of **B**
- F denotes the set of triangles of ***S*** that are traversed by *C*.
- I denotes the set of triangles completely contained in ***D***.
- H denotes the set of hot edges that bound I.
- X denotes the set of triangles of ***S*** stabbed by e_{AB}

The notation **A**|***S***|**B** means that the cells **A** and **B** are fully occluded from each other by ***S***.