MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 838 April, 1985

Prism Trees: An Efficient Representation For Manipulating And Displaying Polyhedra With Many Faces

Jean Ponce

Abstract: Computing surface and/or object intersections is a cornerstone of many algorithms in Geometric Modelling and Computer Graphics, for example Set Operations between solids, or surfaces Ray Casting display. We present an object centered, information preserving, hierarchical representation for polyhedra called *Prism Tree*. We use the representation to decompose the intersection algorithms into two steps: the *localization* of intersections, and their *processing*. When dealing with polyhedra with many faces (typically more than one thousand), the first step is by far the most expensive. The *Prism Tree* structure is used to compute efficiently this *localization* step. A preliminary implementation of the Set Operations and Ray Casting algorithms has been constructed.

© Massachusetts Institute of Technology, 1985

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's Artificial Intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, the Office of Naval Research under contract number N00014-77-C-0389, and the System Development Foundation. This work was done while Jean Ponce was a visiting scientist on leave from INRIA, Paris, France.

1. Introduction

We address the problem of computationally efficient representation of polyhedra. Many algorithms for manipulating polyhedra are naturally decomposed in two steps: the *localization* of the faces that intersect a given field of interest, and the subsequent *processing* of these faces. Here are some practical examples:

1. **Clipping**: *Localize* all the faces of a polyhedron that intersect a viewing pyramid, then *process* them to determine which ones are visible.
2. **Set Operations between solids**: *Localize* all the intersecting faces of two polyhedra, then *process* them to find the intersection polygons of the two polyhedra, and to determine whether or not all the other faces are part of the resulting solid.
3. **Ray Casting**: *Localize* all the faces of a polyhedron intersected by a semi-infinite straight line (the light ray), and *process* these faces to find the first intersection, and compute the display parameters.

In the case of polyhedra with many faces (typically more than one thousand), the *localization* step is the most expensive. This is more generally true of algorithms that manipulate a large number of objects. The basic method proposed in the litterature to speed up this step is to include each object in a simple enclosing box, then build a hierarchy of these boxes, and use it to direct the search. This method has been used for the Clipping and Ray Casting problems by Clark [1976], Rubin and Whitted [1980], and more recently by Weghorst, Hooper, and Greenberg [1985]. It has also been applied to algorithms for Set Operations, essentially in the Constructive Solid Geometry approach (Requicha and Voelcker [1982]).

All these algorithms are "object" based: low level boxes in the hierarchy usually represent either complete polyhedra, or primitive solids. They are not well suited to the manipulation of a set of polyhedra each having thousands of faces. Such polyhedra are of practical importance, in particular they can be models of real, digitized objects that no simple CAD system can handle. Examples are organs from CT scanners (Artzy, Frieder and Hermann [1978]), or objects measured using a laser rangefinder (Faugeras and Pauchon [1983]).

Two recent methods, one for Set Operations between solids (Mantyla and Tamminen [1983]), and the other for Ray Casting of fractal surfaces (Kajiya [1983]) have been designed to deal explicitly with such polyhedra. They use "face" hierarchies that enclose the faces themselves in boxes.

We present a new method, called *Prism Tree*, to represent polyhedra with many faces. This is also a "face" representation, that describes polyhedra as a ternary tree of prisms as enclosing boxes (Figure 1). It generalizes the Strip Tree representation for planar curves (Ballard [1981]) and derives from a polyhedral approximation algorithm (Faugeras et al. [1984]) that is described in Section 2. However, *Prism Trees* have much wider applications, in particular they are *information preserving*, which is not the case of the approximation algorithm.

The representation is intrinsic to the surface, as opposed to the EXCELL structure used by Mantyla and Tamminen [1983], or to the Octree representation for volumes

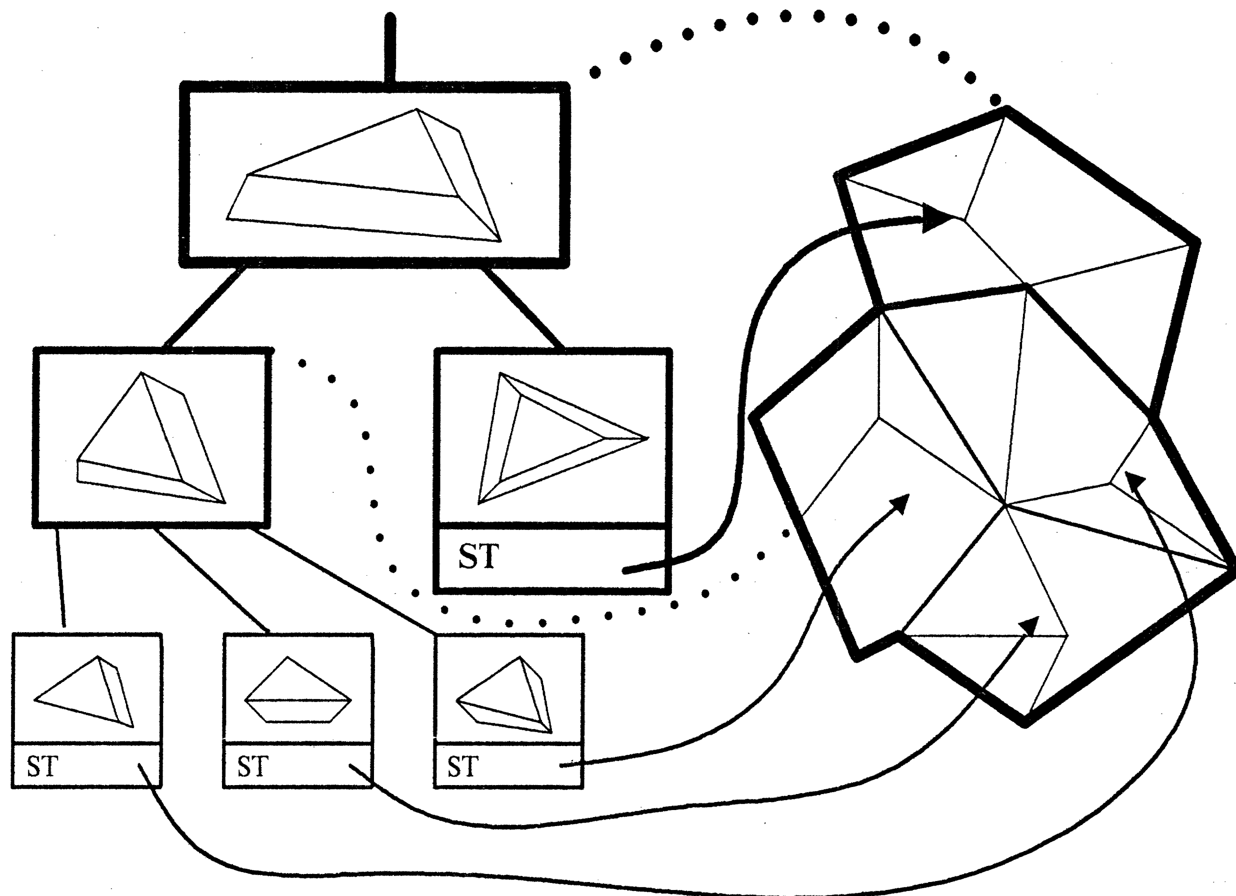

Figure 1. The *Prism Tree* representation for polyhedra. At each level, the surface ST associated with a node is represented by an enclosing prism (dotted lines) The surfaces associated with the sons of a node partition the surface associated to their father. The link between a node and the associated ST is only made explicit at the leaf level (solid lines). As these surfaces partition the original polyhedron, the representation is information preserving.

(Jackins and Tanimoto [1980], Meagher [1982], Iftikhar [1981]), that are attached to a particular reference frame. It is therefore invariant under rigid transformations. We show that Prism Trees can be used to speed up the *localization* step of Set Operations and Ray Casting algorithms for polyhedra with many faces. They unify the approaches of the Mantyla and Tamminen (although the data structures are very different) and Kajiya algorithms, and in general demonstrate an improvement over these methods.

We first sketch the underlying polyhedral approximation algorithm (Section 2). Then we define the *Prism Tree* structure, and give some properties (Section 3). Section 4 describes a set operations algorithms for *Prism Trees*, and Section 5 generalizes to the *Prism Tree* structure a Ray Casting algorithm proposed recently by Kajiya [1983] for fractal surfaces. The crucial *localization* step has been fully implemented. The implementation of the *processing* step is under way. Examples, featuring set operations between synthetic solids and Ray Casting display of both synthetic and real objects, are given. Detailed

procedural versions of the algorithms can be found in the appendix.

2. Polyhedral Approximation

In this section, we describe the polyhedral approximation algorithm (Faugeras et al. [1984]). As we will see in the sequel, the Prism Tree employs a variant of this algorithm. The data consists of a polyhedron of genus 0 (ie without holes, we will see the reason of this restriction below), whose faces are not necessarily triangular. The polyhedron is described by an object graph OG whose nodes are the vertices of the polyhedron, and arcs are the polyhedral edges joining them. The algorithm generalizes to 3D space a recursive polygonal approximation algorithm (Duda and Hart [1973]). Using a breadth first approach, it approximates the initial polyhedron by a polyhedron with triangular faces T_is.

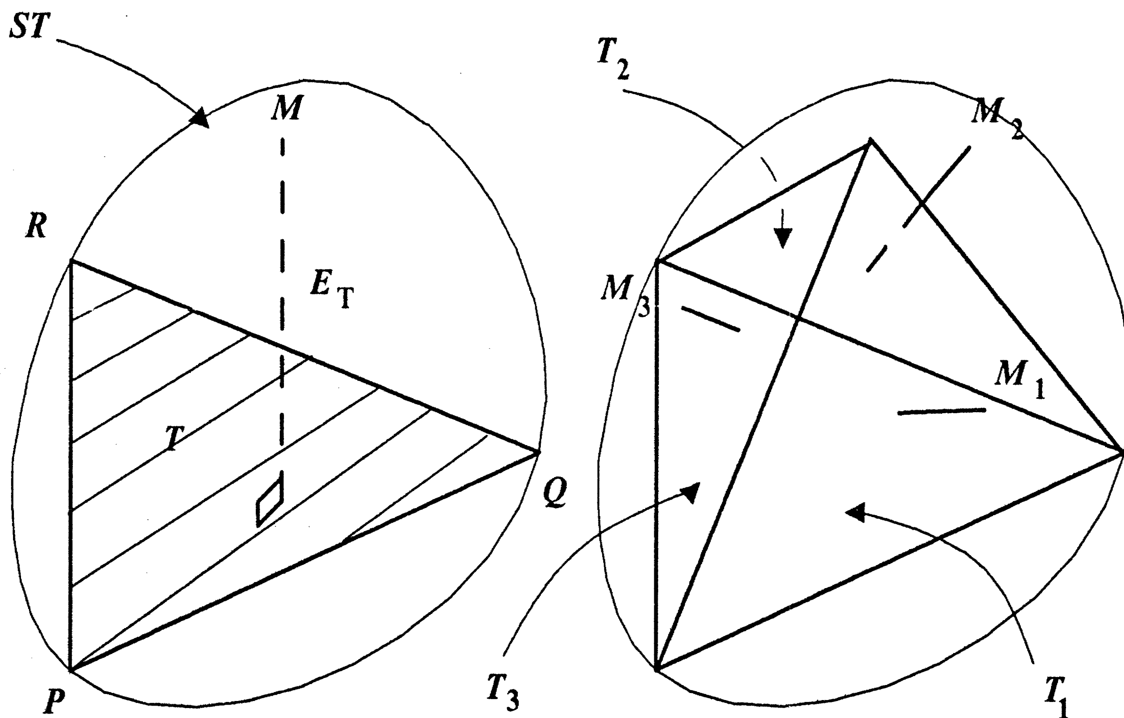

Figure 2. The split step. The triangle $T = PQR$ is replaced by the three triangles $T_1 = PQM, T_2 = QRM, T_3 = RPM$ that are closer to the original surface.

At each level of the recursion, we associate to each triangle T a set of attributes: a part ST of the surface to approximate, the three vertices P, Q, R of T, the error ϵ_T measured by the maximum distance between the points of ST and the triangle plane, and the point M where this maximum is reached. The triangulation is represented by an adjacency graph AG: the nodes are the triangles themselves, and two nodes are linked by an arc iff they share a common edge.

Starting with a rough approximation (described below) of the surface , the algorithm loops over the following steps until the error associated to each triangle is less than a given threshold ϵ.

Split Step: For each triangle T such that $\epsilon_T > \epsilon$, do: (Figure 2)

1. Let T_1, T_2, T_3 be the three triangles PQM, QRM and RPM, and ST_1, ST_2, ST_3 the associated surfaces (obtained with a method to be described), compute the corresponding errors ϵ_{T_1}, ϵ_{T_2}, ϵ_{T_3}, and the points where they are reached M_1, M_2, M_3.
2. Replace T by the T_is in the adjacency graph (in particular, for each previously neighbor of T, replace T in its list of neighbors by the coresponding T_i).

If we only use this step, old edges are never removed, even if they are a very bad approximation of the surface. This is the reason why we use the following step.

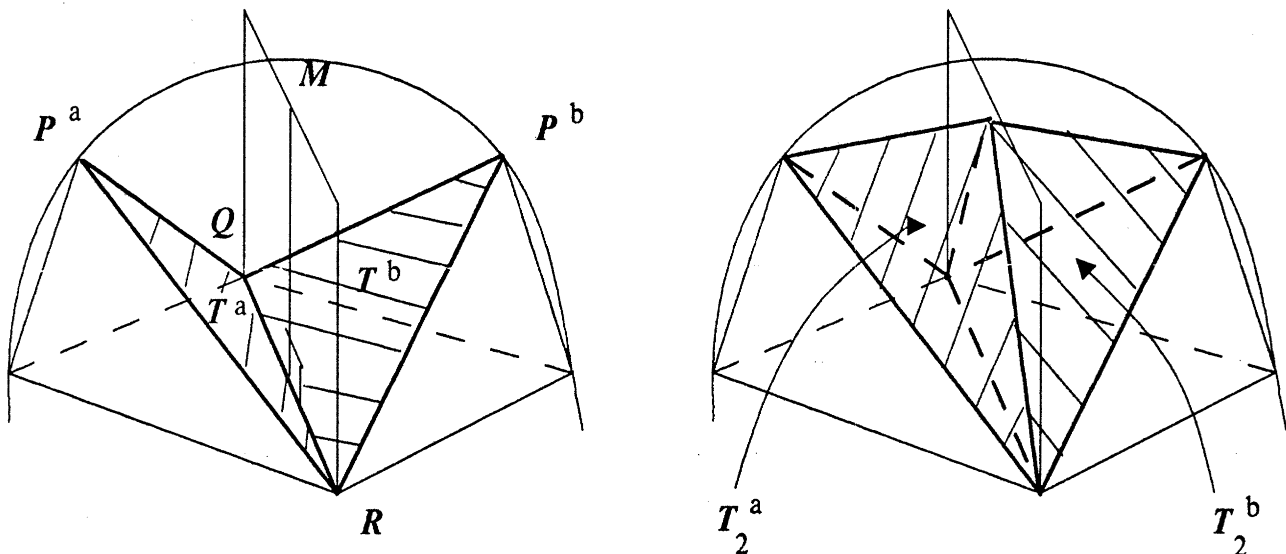

Figure 3. The Adjustment step. The triangles T^a and T^b are replaced respectively by T_1^a and T_2^a, and T_1^b and T_2^b

Adjustment Step: For each pair of neighboring triangles $T^a = P^aQR$ and $T^b = P^bQR$ that have been created at the previous step and whose associated error is greater than ϵ, eliminate the edge QR (Figure 3):

1. Find in $ST^a \cup ST^b$ the point M that is the closest to the bissector plane of T^a and T^b, and lies at the maximum distance from QR.
2. Let T_1^a and T_2^a be the triangles P^aQM and P^aRM, compute as before the associated surfaces ST_1^a, ST_2^a, the errors ϵ_1^a and ϵ_2^a, and the points M_1^a and M_2^a. Compute the same way the triangles T_1^b and T_2^b and their attributes.
3. Replace T^a and T^b by the four new triangles in AG.

We have outlined the algorithm, we now initialize the adjacency graph, and define the STs recursively .

Initialization of AG: Choose three points P, Q, R on the surface (Figure 4). We can compute the shortest cycle of OG that contains these points and is the closest to their plane. As the surface is of genus 0, this cycle cuts it into two connected components ST_1 and ST_2 (Giblin [1977]). The initial graph is composed of two nodes corresponding to the same triangle PQR, but whose associated surfaces are ST_1 and ST_2. The two nodes are linked by three arcs corresponding to the three edges of the triangle. Notice though that any other initial triangulation could have been used. In particular, triangulations based on the smooth patches between surface discontinuities (as in the *Intrinsic Patches* representations of Brady, Ponce, Yuille, and Asada [1984]) could be used to yield better approximations of the original surface.

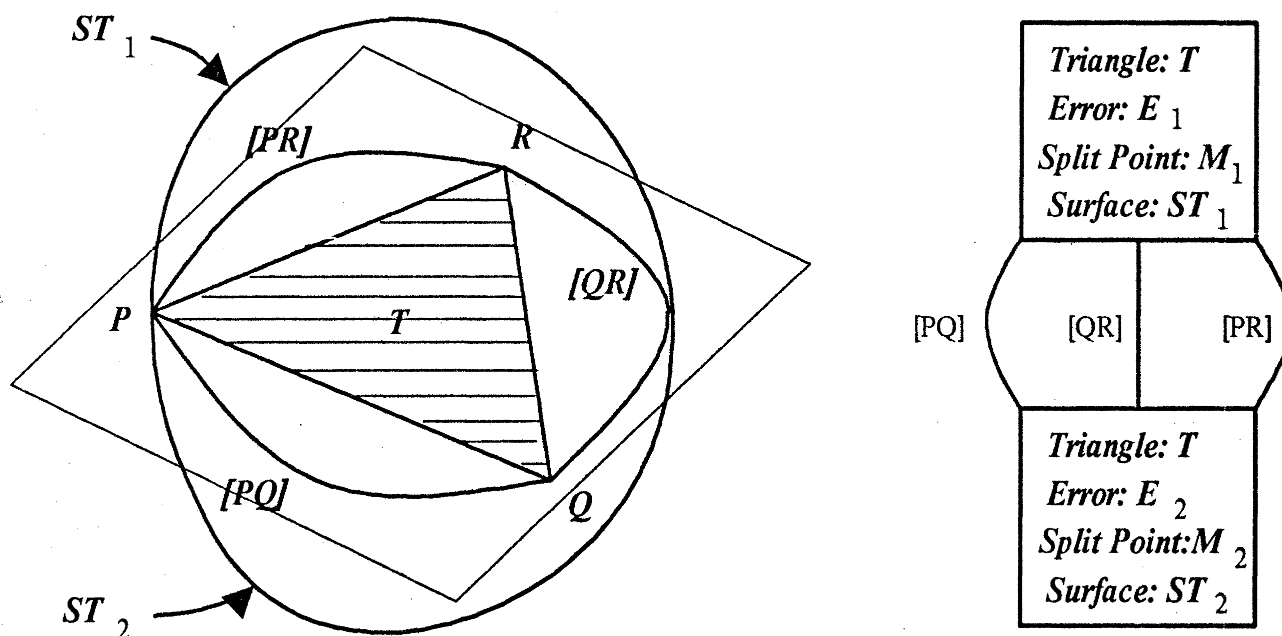

Figure 4. The initial adjacency graph. The three paths [PQ], [QR] and [RP] define a cycle that cuts the surface into two parts ST_1 and ST_2. They define the initial graph.

Definition of the STs: We consider only the split step (the case of the adjustment step is quite similar and will be omitted), and use once again the fact that a cycle of a surface of genus 0 cuts this surface into two components. Suppose (Figure 5) that we have associated to a triangle T a surface ST, delimited by a cycle composed of the three paths $[PQ]$, $[QR]$, and $[RP]$. Let $P_{1,2}, P_{2,3}, P_{3,1}$ be the bissector planes of the pairs (T_1, T_2), (T_2, T_3), and (T_3, T_1). Find a path $[PM]$ between P and M in ST as the shortest path as close as possible to $P_{3,1}$. Define similarly the path $[QM]$. The three paths $[PM]$, $[MQ]$, and $[QP]$ define a cycle $[PMQ]$ in ST that cuts it into two connected components ST_1

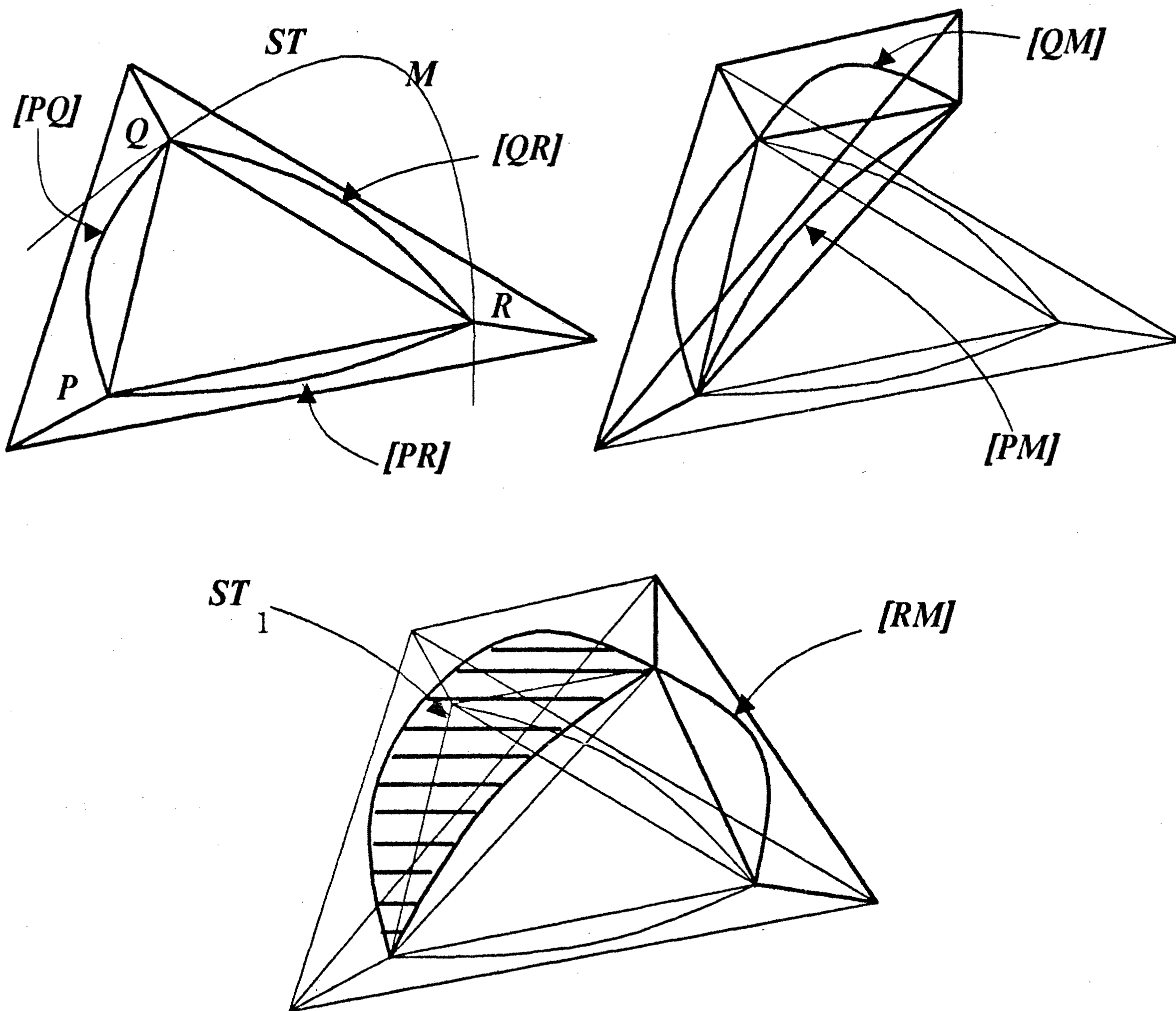

Figure 5. The surface ST is split into three parts by the paths $[PM]$, $[QM]$, and$[RM]$

and ST'. We associate ST_1 to T_1. We can now find in ST' a path $[RM]$, that splits ST' into two components ST_2 and ST_3, that we associate to T_2 and T_3.

The polyhedral approximation algorithm (Faugeras et al. [1984]) was initially proposed in the context of Computer Vision, with the purpose of obtaining from an original polyhedron (typically with 5000 faces) a reasonable approximation with only typically 500 faces. The main point was to compress the information as much as possible, while preserving as much as possible the shape of the object. Unfortunately this is not always possible, in particular for very complicated objects with important concavities, and in practice the approximation can become relatively poor (Ponce 1983]). We now derive from this algorithm the *Prism Tree* structure, and show that these problems disappear, mainly because this representation is *information preserving.*

3. Prism Trees

3.1. Definition

The polyhedral approximation algorithm provides us naturally with a hierarchical representation of the surface: the successive subdivisions of the surface yield a ternary tree structure, where each node represents a triangle and the associated surface, and stores two kinds of information, geometric and structural.

The information carried by the STs is too complex for being directly used at each level of the tree, so the "geometric" part of a non-leaf node is a simple box, easier to manipulate. However, we store in each leaf of the tree a description of the associated connected component ST of OG. In fact this description will not only contain the *edge* information corresponding to OG, but also a description of the *faces* of the original polyhedron adjacent to ST. This is important as it is in fact this *face* information that will be used in the *processing* step of our algorithms.

Since by definition, the STs partition the original graph, *this makes the representation information preserving*: the original polyhedron can be recovered from its *Prism Tree* representation. Consider a triangle T, and the bissector planes P_1, P_2, P_3 of T and its neighbors. We define the box associated with ST to be the smallest truncated pyramid (called prism) with three faces PP_i parallel to the P_is, and the two remaining ones, TB_1 and TB_2, parallel to T, that encloses ST (Figure 6).†

The structural information of each node consists of three pointers, noted $Son[i]$, $i \in [1,2,3]$. The root of the tree points to the two half-surfaces defined by the initialization step. The *Son*s of a node associated to a split triangle point to its three sub-triangles, and the *Son*s of a node associated to an adjusted node point to its two sub-triangles (the third pointer is nil). We will refer to a node by a pointer Pt to it. Figure 7 shows the *Prism Tree* model of a sphere. Notice that the prisms get thinner and thinner as the resolution increases.

3.2. Some Properties

We now give fundamental properties which relate the intersection of two prisms to the intersection of the underlying surfaces. The first one is obvious: if two prisms don't intersect, then the underlying surfaces don't intersect either. In this case, we say that the associated nodes have a *null* intersection.

The converse proposition is evidently false in general: two surfaces can have an empty intersection, although the associated prisms intersect. In fact, it is impossible to decide that the surfaces associated to two nodes intersect before the leaf level, as the STs themselves are only represented at this level. So we say that two nodes that are not

† This is a natural way to define the box as, by definition, ST is the part of the surface that lies within the P_is (or more precisely, it is the part of the surface delimited by a cycle which is the intersection of the surface and the three planes), even though in the general case, points of ST can lie outside of the P_is (remark however that if ST is convex, then it lies entirely inside the P_is). The two remaining faces are used to "close" the box, and their distance corresponds to ϵ_T.

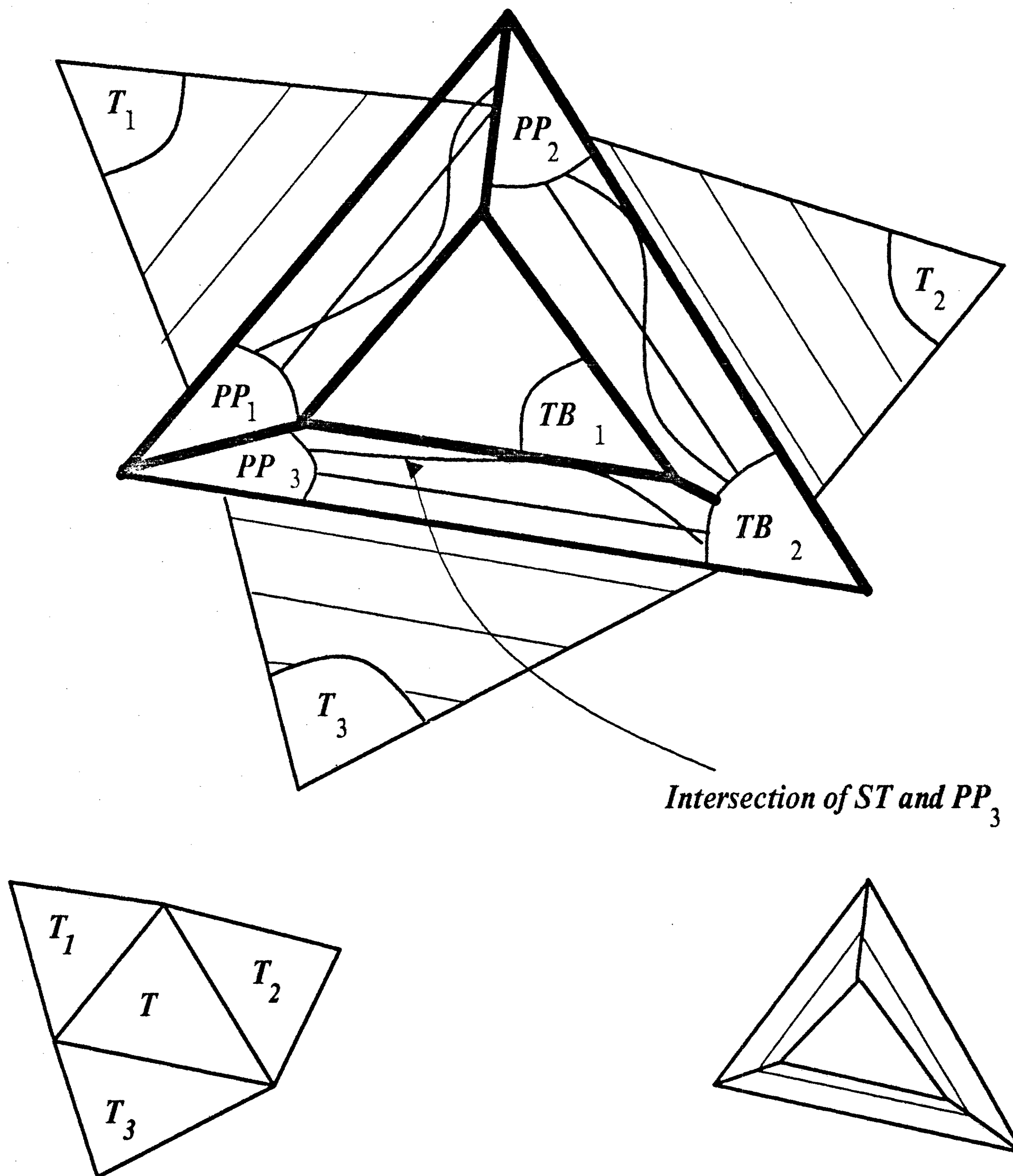

Figure 6. Definition of the prism associated to a triangle. The upper part of the figure shows the geometry of the bissector planes PP_is and the parallel planes TB_is. The lower part shows T and its neighbors T_is, and the constructed enclosing prism.

both leaves have a *possible* intersection. At the leaf node, we can decide if the surfaces associated to two leaves Pt_1 and Pt_2 intersect by testing directly each face of ST_1 against each face of ST_2. If any of these couples intersect, then ST_1 and ST_2 intersect, and we say that Pt_1 and Pt_2 have a *clear* intersection. Otherwise, we say again that they have a *null* intersection.

Let us state two more properties that are obvious, but are the basis of all the algo-

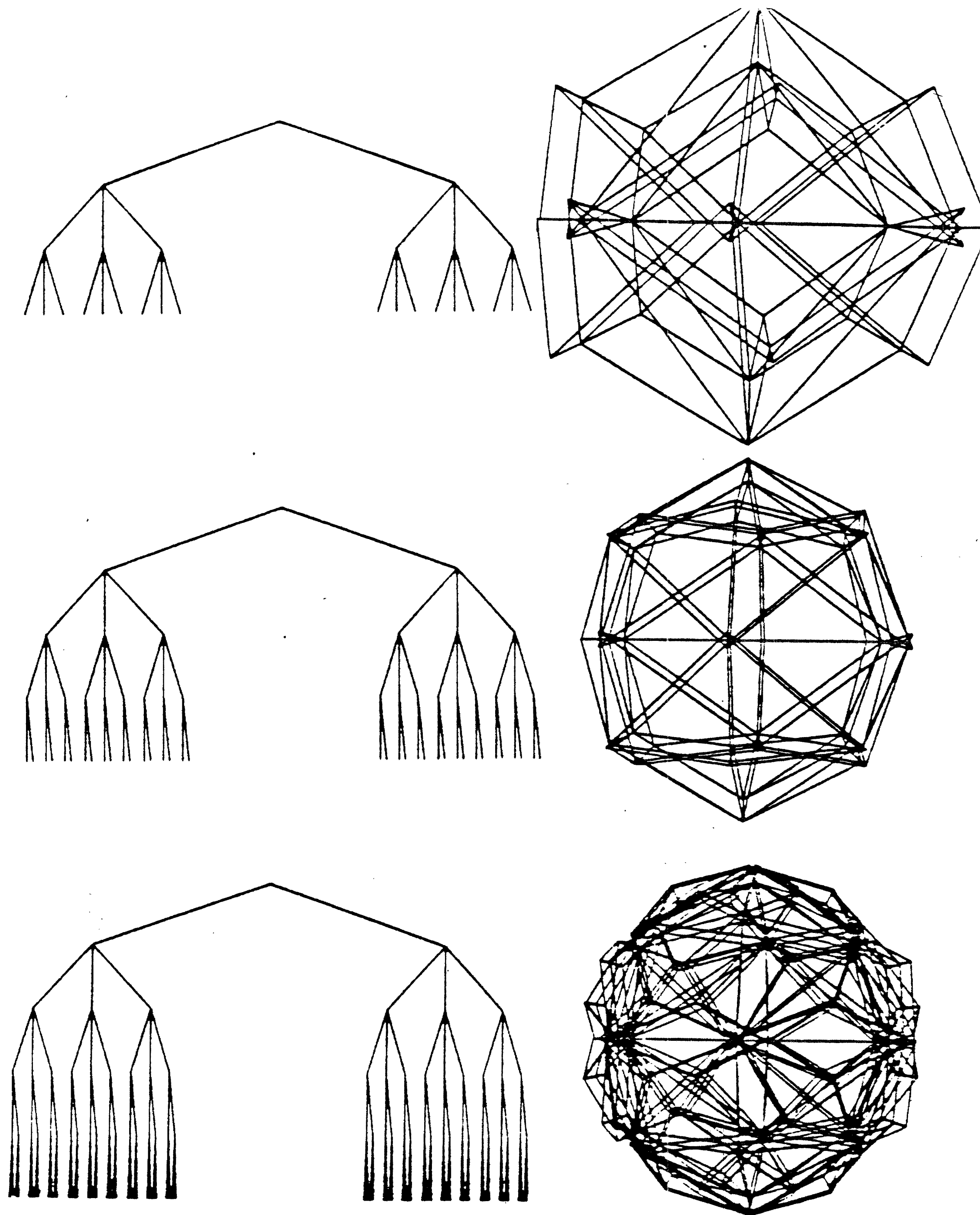

Figure 7. Successive steps in the approximation of a sphere, and the associated Prism Tree

rithms we have developed for manipulating Prism Trees. If the surface associated with a node intersects an object (a line, another surface..), then the same property holds for all the ancestors of the node. Conversely, if the surface associated to a node does not intersect an object, then the same property holds for all the descendants of this node. This will allow the efficient *localization* of intersections by pruning the tree of possibilities as soon as possible.

3.3. A lemma

We have noted in the previous section that the geometric information carried by a non-leaf node of a *Prism Tree* is too poor to directly test the intersection of the underlying surface with another object. We now give a lemma that, given certain additional conditions of regularity, relates the intersection of a straight line and a prism to the intersection of this straight line and the surface ST associated to this prism.

Definition: A prism is said to be *regular* if the associated planes PP_1, PP_2, and PP_3 are the bissector planes P_1, P_2, and P_3 themselves.

The *regular prism* notion is analogous to the notion of *regular strip* introduced by Ballard [1981] for Strip Trees. It ensures that the intersection of the surface ST associated to a regular prism and the union of the PP_is is a closed curve. It is then easy to show the following lemma (see Faugeras and Ponce [1985] for a more general statement and a detailed proof, based on Jordan's theorem).

Clear Intersection Lemma: If a straight line intersects both the planes TB_1 and TB_2 of a *regular* prism, then it intersects the underlying surface ST an odd number of times. As it implies that the line and the surface intersect, we also say that the straight line and the prism have a *clear* intersection (Figure 8).

This lemma could be used for deciding if a point is inside or outside a polyhedron represented by a Prism Tree, by counting the number of *clear* intersections of a semi-infinite straight line traced from this point with the nodes of the tree, using a method similar to the one developed by Kalay [1982] †.

A straight line that has a *clear* intersection with a prism intersects at least one time the associated ST. We have implemented this property in our Ray Casting algorithm (see Section 5) to prune as soon as possible the list of the triangles that may intersect a given ray.

3.4. Some remarks

Contrarily to the polyhedral approximation algorithm, the *Prism Tree* is proposed in the context of Geometric Modelling, and it is used only for *localization* purpose. In our algorithms, the ultimate *processing* step is the same one as for a classical polyhedral representation. This slightly different approach has some important consequences.

First, we use in fact a variant of the approximation algorithm. We no longer stop dividing a node when the error gets small, but when the size (number of points) of the associated ST is less than a given value (6 points per leaf in the examples presented in

† Mantyla and Tamminen [1983], using their Box-EXCELL data structure, have actually implemented a similar method for testing the inclusion of a point inside a polyhedron.

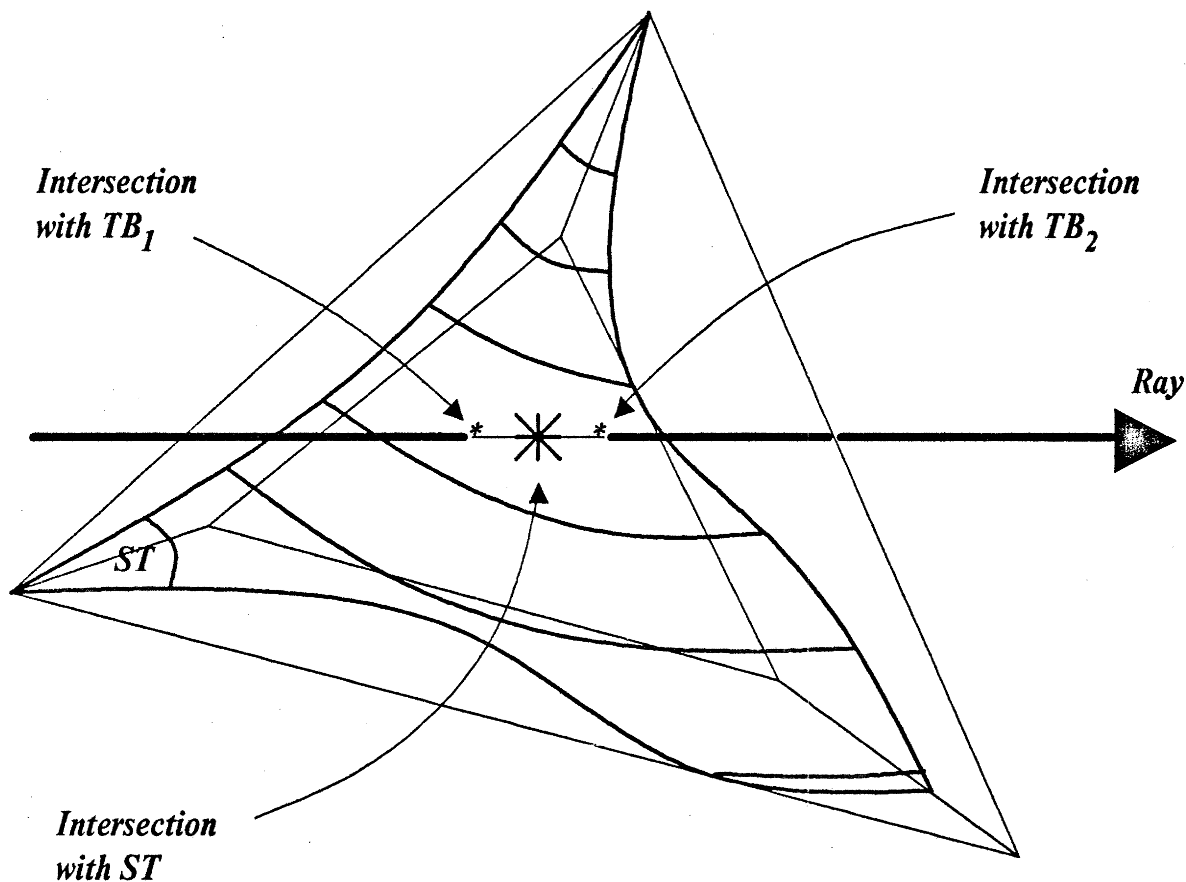

Figure 8. The clear intersection lemma. The ray intersects ST between its two intersections with the TB_is.

---

the sequel): so the depth of the tree representing a n faces polyhedron is $O(Log(n))$, and the number of faces *processed* is proportional to the number of leaves found during the *localization* step.

Second, as the representation is information preserving, and as the *Prism Tree* is used for *localization* only, even poor approximations of an object give exact results. The only difference is that the tree of a poorly approximated object may have bigger prisms at a given level in the tree than a well approximated one, and that the tree can be unbalanced.

Note also that the *Prism Tree* is *intrinsic* to the surface it represents: its geometric features are not constructed according to a particular frame, but to surface features. This implies that the structure of a prism tree is invariant through rigid transformations.

A last, but important remark: the approximation algorithm has been designed for surfaces of genus 0, which is an unfortunate restriction. We can extend our algorithms and analyses to more general surfaces by initially dividing them in components of genus 0 (e.g., cut a torus in two components along its parabolic lines), and considering a simple triangulation of these components as the initial AG. The drawback is the addition of a (possibly) interactive first step to define this triangulation. But as long as this triangu-

lation is simple enough (ie the number of initial nodes of the *Prism Tree* is bounded and small enough), it does not change the overall complexity of the *Prism Tree* algorithms.

The price can be worth paying for including very complicated, real objects in a high level Graphics environment. This is quite similar to the approach advocated earlier, that first segments the surface into smooth patches (Brady, Ponce, Yuille, and Asada [1984]), yielding better approximations of the original surface.

4. Set Operations Between Solids

Boundary representation modellers (Braid [1979], Mantyla and Sulonen [1982]) represent surfaces by faces, edges, and vertices, and the neighborood relations between them. Set operations between solids are then computed by first finding the boundary intersection (which is a set of polygons) of the two objects, and then classifying the faces, edges, and vertices *outside* or *inside* according to the operation considered.

The first step, finding the intersection polygons, is the most expensive one. A naive implementation leads to a complexity of $O(m.n)$, where the two objects compared have respectively m and n faces. To speed up these algorithms, Mantyla and Tamminen [1983] use the *localization/processing* decomposition scheme, by first *localizing* the intersecting faces of the two objects, and then computing the polygons themselves. They organize the 3D space for each polyhedron into a hierarchical data structure, called Box-EXCELL. This structure is not intrinsic to the surface. It consists of a hierarchy of non overlapping rectangular cells, each of them pointing to all the faces whose enclosing boxes intersect it. A cell that intersects too many boxes is subdivided, so that the number of boxes by terminal cells remains small. For each edge of one polyhedron, they find all the faces of the other polyhedron that it intersects by visiting the associated Box-EXCELL structure. Mantyla and Tamminen's experiments show that the *localization* step complexity is reduced to $O(m+n)$ (it cannot be less in their case as each edge of both polyhedra is visited).

We use the same decomposition approach, but the *localization* of the intersection is different. Our algorithm is a direct generalization of the Ballard's algorithm (Ballard [1981]) for finding the intersection of two curves. We represent the two polyhedra by Prism trees and mark all the nodes that correspond to intersecting faces by visiting in parallel the two trees (this is similar to the algorithms for Octrees intersections (Meagher [1980]), although the result obtained by our method will not be approximate, as for Octrees, but exact). At each level, if two prisms don't intersect, then the associated surfaces don't intersect either, so the nodes and all their descendants are marked non-intersecting. Otherwise, if the two nodes are leaves and the associated *ST*s intersect, we mark them as intersecting, as well as their ancestors, otherwise we also mark them non-intersecting. In the remaining case of two non-leaf intersecting prisms, the bigger prism is subdivided (this heuristic, originally proposed by Ballard [1981] in the 2D case, has for purpose to always compare prisms of equivalent sizes), and the recursion proceeds.

The *processing* step then begins: we classify the non intersecting *inside* or *outside* nodes of one tree with respect to the other one by classifying one node, and propagating its status to its connected component of non-intersecting nodes (as two neighboring non-intersecting nodes are necessarily either both *inside* or both *outside*), and eliminate

the *inside* or the *outside* nodes according to the operation performed. The connected component exploration is done by using a neighbor finding algorithm (Faugeras and Ponce [1983,85]), analogous to the one developed by Samet [1982] for Quadtrees, and that finds any neighbor of a node at the same level of the tree in constant average time. We have not yet included in the program the construction of the intersection polygons, but we intend to do it in a near future, by using a solid modeller for polyhedra developed by Lanusse [1985]. As already noted, the addition of this module will not change the overall complexity of the method, as the number of faces by leaf is always bounded.

Figure 9 shows the application of this algorithm to set operations between a sphere and a double cone, whose Prism Trees representations contain approximatively 1000 leaves each. The computing time for the *localization* step is only $25s$ on a VAX class minicomputer. Of the million of possible intersections, less than one thousand are actually tested. The processing step takes about $1s$.

We may make some remarks about the complexity of the localization procedure. At a given level of the tree, only the intersecting nodes are again subdivided. This would lead, if only one of the sons of each intersecting node was also an intersecting one, to a complexity of $O(N_i.K)$, where N_i is the number of intersecting faces, and K the depth of the tree. Such a best case is evidently not realized in practical experiments, but it indicates that the complexity depends on the shape of the intersection polygon, and on the logarithm of the total number of faces (corresponding to the depth K of the tree), so the method seems better than the Mantyla and Tamminen one, although a more careful study is necessary. Notice anyway that an advantage of this method is that, the representation being intrinsic, it does not have to be recomputed when the object is rotated (the same advantage holds with respect to Octrees (Iftikhar [1981]). A procedural version of the *localization* algorithm is given in the appendix.

5. Ray Casting

Hidden surface elimination algorithms (Sutherland, Sproull, and Schumacker [1974]), and in particular Ray Casting methods (Roth [1982]), although they are among the more general and allow a lot of "special effects", are usually computationally expensive: if N_s is the number of surface elements, N_s ray-surface intersections must be computed for each pixel. This makes the "naive" version of this approach unusable for moderately complex polyhedral objects each often containing more than 1000 facets for images containing at least a quarter million pixels. To increase the efficiency of Ray Casting algorithms, *localization* methods have been used.

Clark [1976], and after him Rubin and Whitted [1980], enclose the objects composing the scene in a hierarchy of simple boxes (parallelepipeds). At each level, if the box does not intersect the ray, the corresponding objects are rejected. Otherwise, the box is subdivided, etc.. Weghorst, Hooper, and Greenberg [1985] have successfully applied more sophisticated versions of similar methods to very complex scenes. However, these methods are *object oriented*: they use hierarchies of objects, bounded by simple volumes, but they are not really designed for dealing with polyhedra that contain themselves thousands of faces.

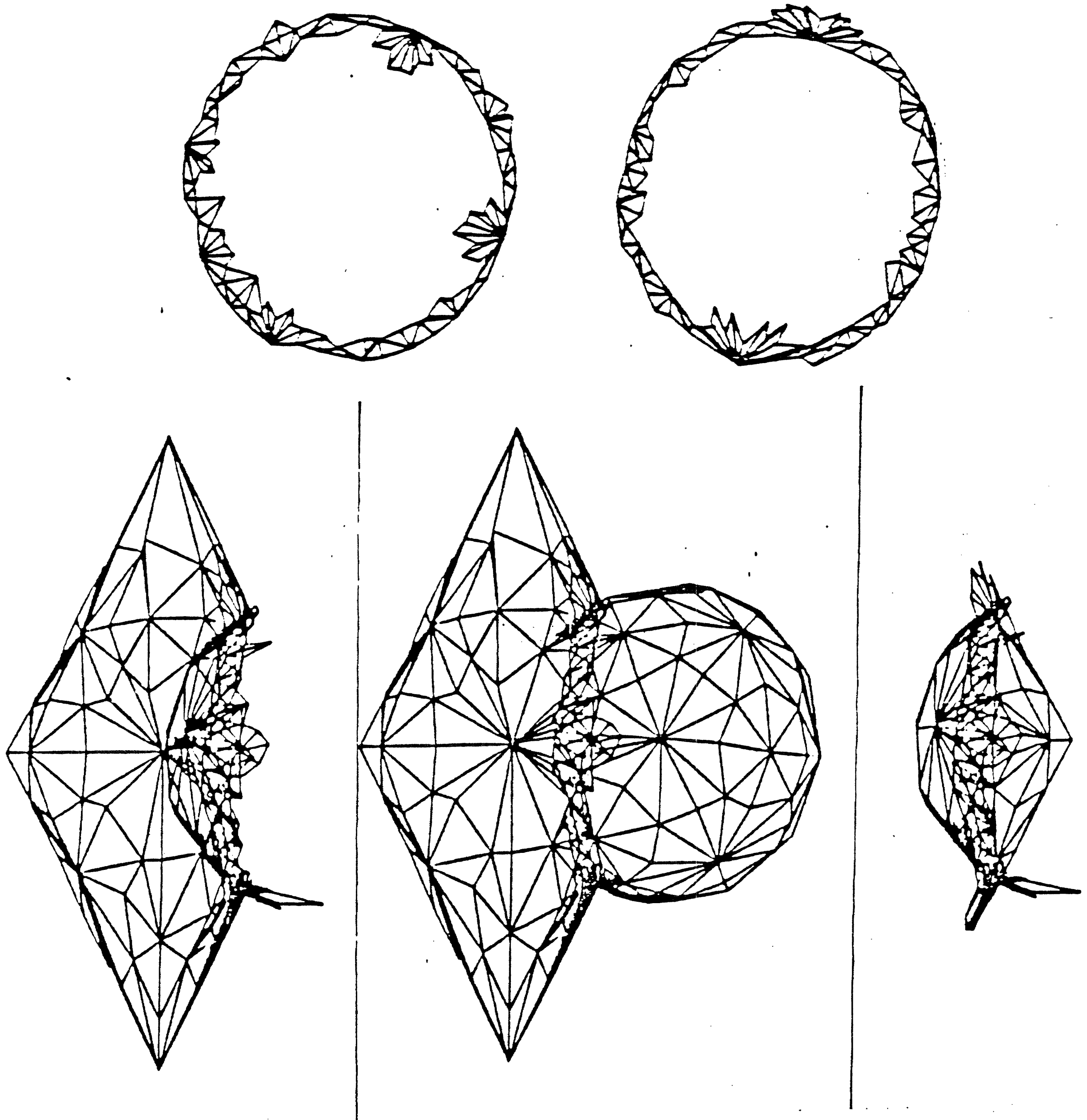

Figure 9. The upper part of the figure shows the leaves marked intersection in the trees representing the double-cone and the sphere. The lower part shows the difference, union, and intersection of the two objects

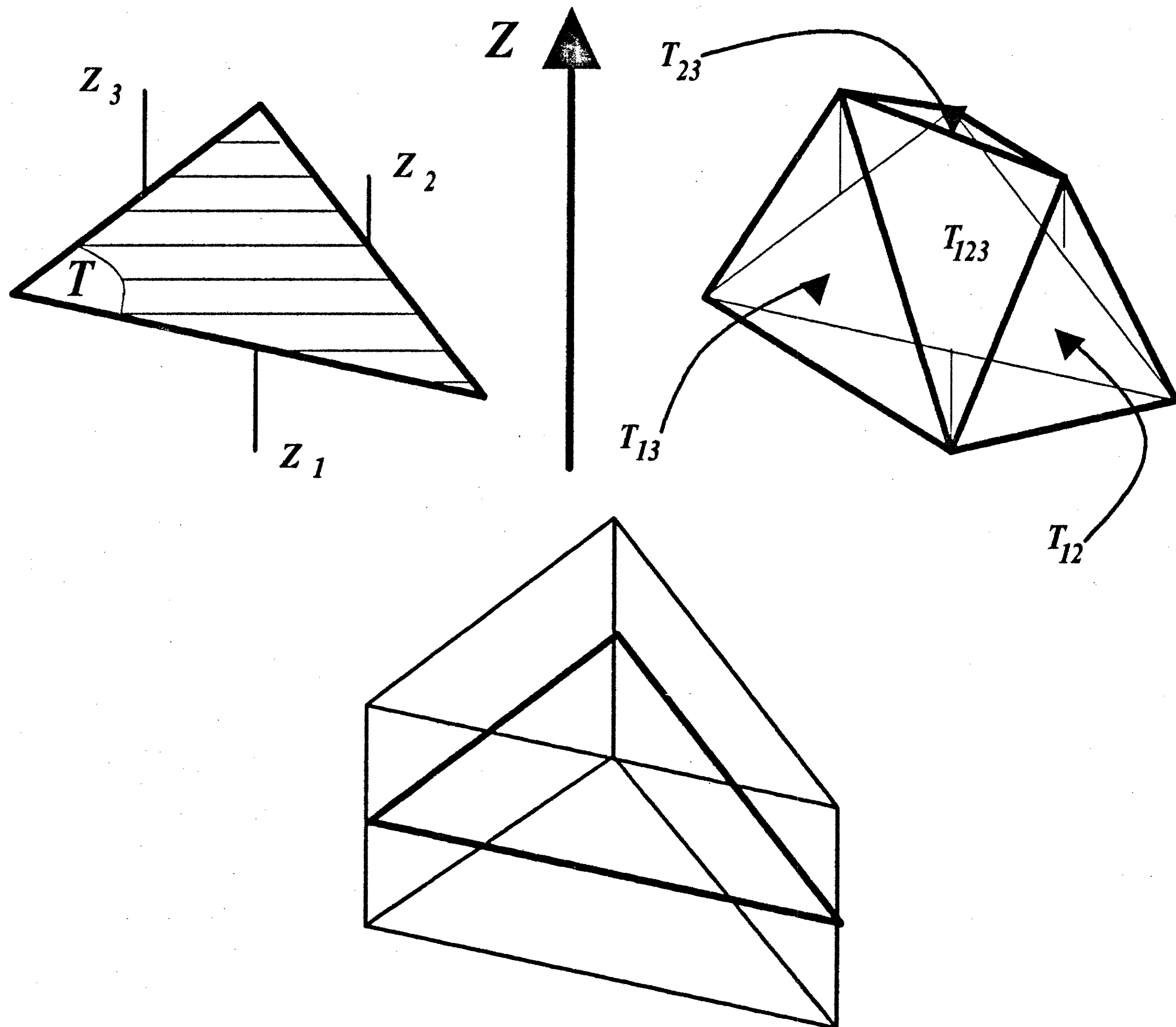

Figure 10. The fractal subdivision process: for each edge of the original triangle T, a subdivision point of height z_i is computed, and T is replaced by the four triangles T_{123}, T_{12}, T_{23}, and T_{13}. The "cheesecake" is obtained by translating the original triangle along the Z axis.

Recently, Kajiya [1983] has proposed a *face oriented* method for rendering fractal surfaces (Fournier, Russel and Carpenter [1982]) using Ray Casting. His surface model is similar to ours, except that it is non deterministic. The surface is a polygonal height field. It is recursively subdivided, each triangle being replaced by four subtriangles whose vertices are generated by a stochastic process (the subdivision in 4 triangles eliminates the need for an adjustment step, but on the other hand, the subdivision cannot be stopped for some triangles as the polyhedral structure would not be preserved). Kajiya encloses each triangle in a so called "cheesecake" extent (Figure 10), which is a prism obtained by translating the triangle along the vertical to enclose the associated surface. To intersect the ray with a surface, Kajiya visits the associated tree: if the ray intersects a box, then it

is again tested against the descendants. Otherwise, the ray is guaranteed not to intersect any of the descendants. The tree is visited until the first intersecting leaf triangle is found.

The method is readily extendible to *Prism Trees.* There are, however, some differences: as the fractal surface is defined as a height field, the cheesecakes associated to the sons of a given node never intersect each other. This implies that they can always be sorted from the nearest to the farthest with respect to a given ray. In particular, if only one fractal surface is displayed, the corresponding tree can be visited in a strict depth first manner. If the closest node is always visited first, then it is guaranteed that the first intersection found will be the actually closest one.

Prism Trees do not have this nice property, and in particular *non-regular* sons of a same node may intersect. We have to maintain an active list of all the nodes whose associated surface may intersect a given ray. This disadvantage is a price to pay for generality. Notice however that the same problem would arise for fractal surfaces defined on non-planar objects, or simply when several of them are to be displayed simultaneously. Moreover, the *clear intersection lemma* of Section 3 is going to help us keeping the active list short.

Assume that the ray is parameterized by λ, and let $\lambda_{min}(Pt)$ and $\lambda_{max}(Pt)$ be the values of this parameter at the extremities of the intersection of the ray with the node Pt. Following Kajiya, we say that a node Pt_1 shadows a node Pt_2 if $\lambda_{max}(Pt_1) < \lambda_{min}(Pt_1)$ (Figure 11). Kajiya conjectures that this notion could be used to prune the active list, but remarks that, unfortunately, a node can be shadowed at a given level and though have visible descendants, as the intersection of the ray with the box does not imply the intersection of the ray and the surface. We solve this problem by proving the following lemma.

Shadowing Lemma: if the intersection of *Ray* and Pt_1 is *clear*, and if Pt_2 is shadowed by Pt_1, then Pt_2 and his descendants are not visible from the considered pixel, and Pt_2 can be removed from the active list.

Proof: as the intersection is *clear*, *Ray* intersects the surface associated to Pt_1 at a point that verifies $\lambda_{min}(Pt_1) \leq \lambda \leq \lambda_{max}(Pt_1)$, and so hides the surface associated to Pt_2, whose each point verifies $\lambda_{max}(Pt_1) < \lambda_{min}(Pt_2) \leq \lambda$ ∎

Using this property, we can find for each pixel the point of the scene seen by the pixel by a breadth first visit of the trees of the objects composing the scene (so as to eliminate as early as possible the shadowed nodes). At each step of the recursion a λ_{min}-sorted list of the nodes that intersect the ray at this level is maintained. The process terminates when the list is either empty or only composed of leaves. The first element of the list is then used to compute the display parameters, by finding among the (few) faces associated to this leaf the closest one that intersects the ray.

Again, we discuss the complexity of this algorithm. First notice the interest of the *clear* intersection notion. In the ideal case, there are only *clear* and *null* intersections, so the algorithm visits only one branch per tree and per pixel, and due to the shadowing between trees, one can expect to visit entirely a single branch of a single tree. This is an important advantage over Kajiya's algorithm, where the execution time is proportional

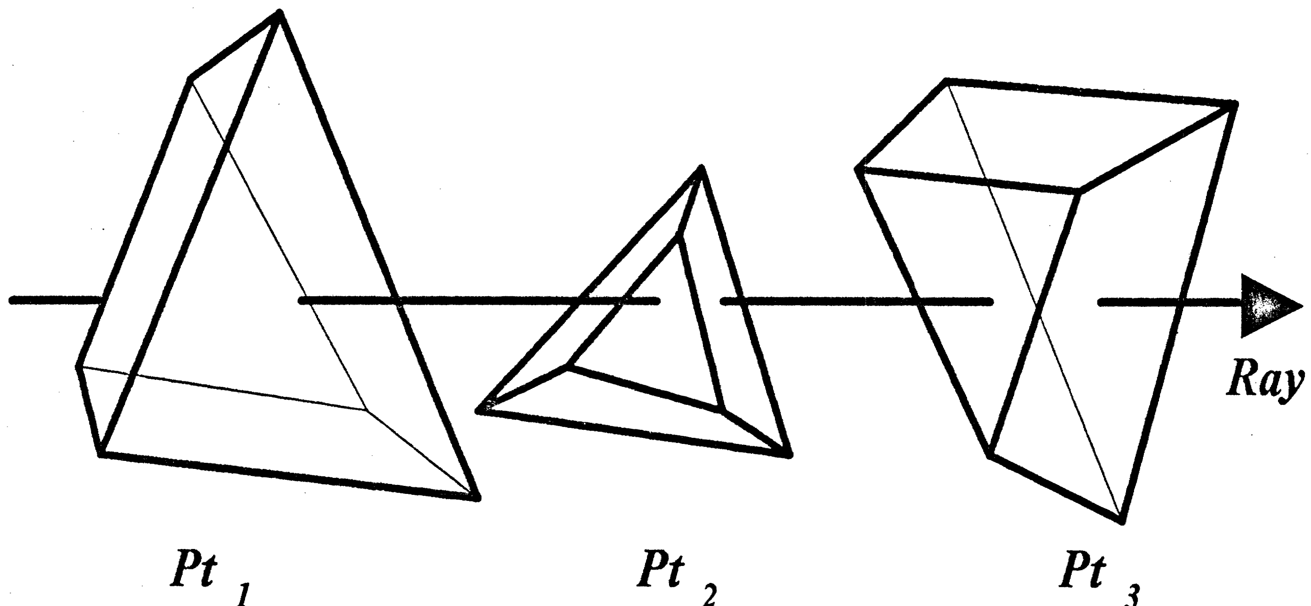

Figure 11. The intersection of a ray with the nodes of a *Prism Tree*. Pt_1 and *Ray* have a *clear* intersection, so Pt_2 and Pt_3, that it shadows, have no visible descendants. Conversely, the shadowing of Pt_3 by Pt_2 alone would not imply that Pt_3 has no visible descendant, as Pt_2 and *Ray* don't have a *clear* intersection.

to the number of objects in the scene (although it is unlikely that several fractal surfaces are going to be displayed in the same image).

In the case where few *clear* intersections occur, or the different surfaces are close enough so that only few shadowings occur, the complexity degenerates, and becomes again proportional to the number of objects. Even in this case though, it is likely that a ray will in general intersect only one of the sons of a node, so only a few branches will be explored.

Figure 12 shows the application of this algorithm to the union of the double cone and the sphere, and to a Renault automobile part. The resolutions of the images are respectively 512×512, with a CPU time of two hours for 2000 leaves, and 256×256, with a CPU time of 28 minutes for 1000 leaves. Once again, in our case, the *processing* step is not complete, as we display the approximation and not the surface itself. Nevertheless, the remark made at the end of the previous section still holds, and a complete implementation will not change the overall complexity of the algorithm. A procedural version of the algorithm can be found in the appendix.

6. Conclusion

We have presented a new method for localizing the search for geometric intersections in the polyhedral case. It has proved efficient for the two classical problems studied. We are continuing our work on the complete implementation of the processing step. Also, a more rigorous study of the algorithms complexity is needed.

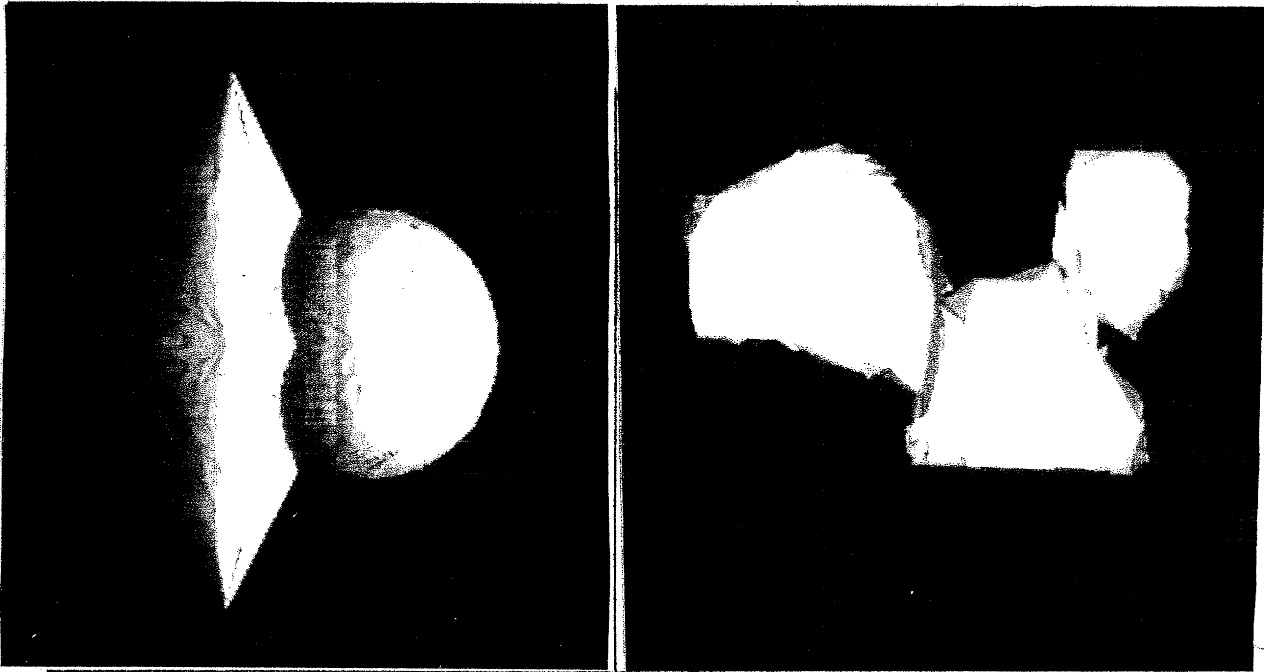

Figure 12. Examples of applications of Ray Casting to the union of the double cone and the sphere, and to an automobile part

We believe that *Prism Trees* have important potential applications, in particular for complex, real 3D objects, digitized as polyhedra with thousands of faces†. In this optic, we give two examples.

Today's most sophisticated Graphics systems deal with (relatively) simple objects. Using the Ray Casting algorithm in conjonction with a laser rangefinding system (Faugeras and Pauchon [1983]) will allow us to include "real life" objects, from industrial parts to statues, in such an high level Graphics environment and to obtain from them the realistic images that only Ray Casting programs authorize.

Similarly, the Set Operations algorithm could be used in CT tomography (Artzy, Frieder and Hermann [1978]) to obtain slices of an organ in an arbitrary direction, or more generally to cut it along an arbitrary surface.

7. Appendix

We now present in more detail the procedures used to find the intersection of two surfaces, and to display the object represented by the Prism Tree using a Ray Casting method. We give them in a pseudo PASCAL form, the declarations of types and variables being omitted.

7.1. The Surface-Surface Intersection Procedure

This procedure is straightforward, the two trees are visited in parallel in a depth first search manner, until all the intersection leaves are marked (using the flag *Mark*).

† This is the typical output of scanning processes that measure successive slices of the surface of a 3D object, as laser digitizers or CT scanning.

Then all their ancestors are also marked. We suppose that we dispose of the following functions:

Vol (Pt): this real function returns the volume of the box associated to Pt.

Leaf (Pt): this boolean function returns *true* iff the node Pt is a leaf.

Marked (Pt): this boolean function returns *true* iff Pt is not *nil* and is marked (i.e. the flag *Mark* is set to 1).

Prismint (Pt_1, Pt_2) is a 3-valued function that returns *clear*, *possible*, or *null* according to the type of intersection of Pt_1 and Pt_2. This function is the key of the algorithms that manipulate the Prism Trees. In the special case of the leaves whose associated prisms intersect (and only in this case), *Prismint* tests directly the intersection of the surface patches themselves, and returns *clear* if they actually intersect.

```
Procedure Find - Surfaces - Intersection (Pt1,Pt2)
begin
if Prismint (Pt1,Pt2) ≠ null then
      begin
      Le1 ← Leaf (Pt1); Le2 ← Leaf (Pt2);
      if Le1 and Le2
            then begin Pt1 ↑ .Mark ← 1; Pt2 ↑ .Mark ← 1 end
            else (* The recursion proceeds *)
                  if Le2 or (Vol (Pt1) > Vol (Pt2))
                        then begin
                              for i ← 1 to 3 do
                                    Find - Surfaces - Intersection (Pt1 ↑ .Son [i], Pt2);
                              if Marked (Pt1 ↑ .Son [1])
                              or Marked (Pt1 ↑ .Son [2])
                              or Marked (Pt1 ↑ .Son [3])
                                    then Pt1 ↑ .Mark ← 1
                              end
                        else begin
                              for i ← 1 to 3 do
                                    Find - Surfaces - Intersection (Pt1, Pt2 ↑ .Son [i]);
                              if Marked (Pt2 ↑ .Son [1])
                              or Marked (Pt2 ↑ .Son [2])
                              or Marked (Pt2 ↑ .Son [3])
                                    then Pt2 ↑ .Mark ← 1
                              end
      end
end;
```

7.2. The Ray Casting Procedure

The elements of the λ_{min}-sorted list of nodes, *Active – List*, are 5-fields records: the first field, *Node*, is the corresponding node of the Prism Tree, the second one, *Next*, is a pointer to the next element of the list, the third one, *Old* is used for the breadth first visit of the tree. Its value is 1 if the node has been inserted at the previous iteration. The last two fields caracterize the intersection of the ray with the prism. λ_{Node} is the

minimum value of the parameter λ of the ray such that an intersection occurs, and P_{Node}, set only if the node is a leaf and has a *clear* intersection with the ray, is the firstpoint of intersection of ST and the ray. It is used for computing the display parameters.

We suppose to dispose of the following procedure and function:

Insert (Q, N, λ_{min}) inserts the element Np in the list Q by comparing the λ_{min} value associated to N to the one of the currently visited node.

Rayint $(Ray, Pt, \lambda_{min}, \lambda_{max}, P_{min})$ is a modified version of *Prismint*. It is also a 3-valued function, the *clear* intersection notion being extended to the intersection of a straight line and a regular prism. Its additional features are that it returns the extremal values λ_{min} and λ_{max} of the intersection of Ray and Pt, and in the case where Pt is a leaf and the intersection is *clear*, it also returns the closest intersection point P_{min}.

```
Function Raycast (Ray, Pt, P_seen):boolean;
begin (* initialize parameters *)
with Active - List ↑ .Next ↑ do
      begin Node ← Pt; Old ← 1; λ_Node ← -∞ end;
λ_max,0 ← +∞; λ ← +∞; P_seen ← nil;
repeat (* main loop *)
      All - Leaves ← true; Empty ← true;
      P ← Active - List; Q ← P ↑ .Next;
      while Q ≠ nil do (* auxiliary loop: visit the list *)
            if Q ↑ .λ_Node > λ_max,0
                  then Q ← nil (* the rest of the list is shadowed *)
            else if Leaf (Q ↑ .Node) (* non shadowed leaf? if yes, it shadows *)
                        then begin Empty ← false; Q ← nil end (* everything behind *)
                        else if Q ↑ .Old = 0
                              then (* non-leaf new node that gets old, go on *)
                                    begin All - Leaves ← false; Q ↑ .Old ← 1;
                                    P ← Q; Q ← Q ↑ .Next end
                              else begin (* non leaf old node, divide *)
                                    for i ← 1 to 3 do
                                          begin
                                          Inter ← Rayint (Ray, Q ↑ .Node ↑ .Son [i], λ_min, λ_max, P_min);
                                          if (Inter ≠ null) and (λ_min ≤ λ_max,0) then
                                                begin (* insert the son *)
                                                New(Np); Insert (Q, Np, λ_min);
                                                with Np ↑ do
                                                      begin Old ← 0; λ_Node ← λ_min; P_Node ← P_min;
                                                      Node ← Q ↑ .Node ↑ .Son [i] end;
                                                if (Inter = clear) and (λ_max < λ_max,0)
                                                      then λ_max,0 ← λ_max;
                                          end end;
                                    Q ← Q ↑ .Next; P ↑ .Next ← Q
                                    end;
until All - Leaves or Empty;
if Empty then Raycast ← false
      else begin P_seen ← Active - List ↑ .Next ↑ .P_Node; Raycast ← true end
end;
```

References

1. Artzy, E., Frieder., G., Hermann, G.T., The theory, design, implementation, and evaluation of a 3D surface detection algorithm, Computer Graphics, Vol 12 (1978), p. 153-160.
2. Ballard, D.H., Strip Trees: A hierarchical representation for curves. Comm. of the ACM, Vol 24, No 5 (May 1981), p. 310-321.
3. Braid, I.C., Notes on a geometric modeller, CAD Group Doc. 101, Univ. Cambridge, Cambridge, England (June 1979).
4. Brady, J.M., Ponce, J., Yuille, A., and Asada H., Describing surfaces, Proceedings of the 2nd Int. Symp. Rob. Res., Kyoto, Japan, Hanafusa, H., and Inoue, H., (eds.), MIT Press, 1985. (also in Computer Vision, Graphics, and Image Processing, August 1985)
5. Clark, J.H., Hierarchical geometric models for visible surface algorithms, Comm. of the ACM, Vol 19, No 10 (October 76), p. 547-554.
6. Doctor, L., and Torborg, J., Interactive solid modelling design station utilizing Octree encoding, Rensselaer Polytechnic Institute, Techn. Report 80010 (1981).
7. Duda, R.O., and Hart, P.E., **Pattern classification and scene analysis**, Wiley Interscience, New York (1973).
8. Faugeras, O., Hebert, M., Mussi, P., and Boissonnat, J.D., Polyhedral approximation of 3D objects without holes, Comp. Gr. Im. Proc. (1984).
9. Faugeras, O., and Pauchon, E., Measuring the shape of 3D objects, Proc. CVPR, Washington D.C. (June 1983), p. 2-7.
10. Faugeras, O., and Ponce, J., Prism Trees: a hierarchical representation for 3D objects, Proc. IJCAI-83, Karlruhe (August 1983), p. 982-988.
11. Faugeras, O., and Ponce, J., An object centered representation for 3D objects: the Prism Tree, submitted to Comp. Gr. Vision and Im. Proc. [1985]
12. Fournier, A., Fussel, D., and Crapenter, L., Computer rendering of stochastic models, Comm. of the ACM, Vol 25, No 6 (June 1982), p. 371-384.
13. Giblin, P.J., **Graphs surfaces and homology**, Chapman and Hall, New York (1977).
14. Hunter, G.M., Efficient computation and data structures for graphics, Ph.d diss., Department of electrical engineering and computer science, Princeton Univ., Princeton, N.J. (1978).
15. Iftikhar, A., Linear geometric transformations on Octrees, TR-81-003, Rensselaer Polytechnic Institute, Troy, New York (May 1981).
16. Jackins, C.L., and Tanimoto, S.L., Oct-trees and their use in representing 3D objects, Dept. of Comp. Science, Univ. of Washington, Seattle, Techn. Rep. 79-07-06 (1980).
17. Kajiya, J.T., New techniques for ray tracing procedurally defined objects, Proc. SIGGRAPH 83 (july 1983), p. 91-102.

18. Kalay, Y.E., Determining the spatial containment of a point in general polyhedra, Comp. Gr. Image Pr., Vol 19 (1982), p. 303-334.

19. Lanusse, A., A general algorithm for intersecting polyhedra, MIT AI Memo, in preparation [1985].

20. Mantyla, M., and Sulonen, R., GWB, a solid modeller with Euler operators, IEEE Computers Graphics and Applications, Vol 2, No 7 (1982), p.17-31.

21. Mantyla, M., and Tamminen, M. ,Localized set operations for solid modelling, Proc. SIGGRAPH 83 (july 1983), p. 279-288).

22. Meagher, D., Geometric modelling using octree encoding, Comp. Gr. Image Pr., Vol 19 (1982), p. 129-147.

23. Ponce, J., Representation et Manipulation d'Objets Tridimensionnels, PhD Thesis, University of Orsay, Paris, France, 1983.

24. Requicha, A.A.G., and Voelcker, H.B., Solid modeling: a historical summary and contemporary assessment, IEEE Comp. Graphics and Appl., Vol 2, No 2 (1982), p. 9-24

25. Rubin, S.M., and Whitted, T., A 3D representation for fast rendering of complex scenes, Computer Graphics, Vol 14, No 3 (August 1980), p. 110-116.

26. Roth, S.D., Ray Casting for modelling solids, Comp. Gr. Image Pr., Vol 18 (1982), p. 109-144.

27. Samet, H., Neighbor finding techniques for images represented by quadtrees, Comp. Gr. Image Pr., Vol 18 (1982).

28. Sutherland, I.E., Sproull, R.F., and Scumacker, R.A., A characterization of ten hidden surface algorithms, Comput. Surveys, Vol. 6 (1974), p. 1-55.

29. Tanimoto, S., and Pavlidis, T., A hierarchical data structure for picture processing, Comp. Gr. Image Pr., Vol 4, No 2 (1975).

30. Tilove, R.B., Set membership classification: a unified approach to geometric intersections problems, IEEE Trans. on Comp., C 29 (10) (1980), p. 874-883.

31. Warnock, J.E., A hidden line algorithm for halftone picture representation, Techn. Rep. TR4-15, Comp. Sci. Dept., Univ. Utah (1969).

32. Weghorst, H., Hooper, G., Greenberg, D.P., Improved computational methods for Ray Tracing, ACM Trans. on Comp. Graphics, Vol 3, No 1 (January 1985), p. 52-69.