Rapportnr.: 88-20 July 1988

Rijksuniversiteit te Leiden

Vakgroep Informatica

The creation and display of arbitrary polyhedra in HIRASP

Wim J.M. Teunissen

Peter J.M. van Oosterom

Department of Computer Science
Leiden University
Niels Bohrweg 1
P.O. Box 9512
2300 RA Leiden
The Netherlands

THE CREATION AND DISPLAY OF ARBITRARY POLYHEDRA IN HIRASP

Wim J.M. Teunissen* and Peter J.M. van Oosterom†

Department of Computer Science, University of Leiden,
P.O. Box 9512, 2500 RA Leiden, The Netherlands,
Email: oosterom@hlerul5.bitnet.

1 Introduction.

Hirasp (Hierarchical Interactive Raster graphics System based on Patterns) is a general purpose 3D-modeling system for raster graphics devices [1]. It contains and combines several new concepts, such as: patterns defined by a domain and a colour function, pattern expressions, and pattern graphs. Apart from 0D, 1D and 2D-domains (e.g. marker, line and polygon) Hirasp also provides 3D-domains. During the implementation (written in C for a Commodore Amiga) of some of the 3D-aspects of Hirasp we encountered some interesting problems. New solutions were found for the following subproblems:

1) A "Point in Polyhedron"-test is needed to determine whether the eye-position is inside the polyhedron. In that case the polyhedron is not displayed; see section 4.
2) Make the normal vectors of the faces which describe the polyhedron, point in the right direction, i.e. away from the polyhedron; see section 5.
3) A fast method for checking whether a set of faces represents a correct polyhedron; see section 6.

Creating a polyhedron is described in section 2. In section 3 we discuss the BSP-tree, the data structure used to implement hidden-surface removal.

2 Creating a polyhedron.

One of the 3D domain functions in Hirasp generates polyhedra. A polyhedron may be defined by either a set of points (yielding the convex hull), or by a set of polygons representing the faces of the polyhedron. The first definition always results in a convex polyhedron which can be displayed relatively easy. The latter definition requires checking whether the set of polygons represents a correct 3-dimensional object. If so, the BSP-tree is built, and the normal vectors of the faces are made to point in the right direction, thus allowing fast generation of an image of the polyhedron with its hidden surfaces removed.

The following restrictions are imposed on the input data when creating a polyhedron: the faces (polygons) must be planar and the edges of a face may not cross, but a face may be concave and there is no restriction on the number of edges. A polyhedron must be bounded: a polyhedron with missing or dangling faces is not accepted. It may however consist of several (unconnected) parts. To check if a given set of polygons meets these criteria a rather time consuming search has to be performed. During this search the faces that are *neighbours* of each other are recorded.

3 The BSP-tree.

The datastructure we chose to allow hidden surface elimination is the Binary Space Partitioning (BSP) tree [2]. Major advantage of the BSP-tree is that, once built, it allows the generation of an image with

---

* Supported by the Dutch Technology Foundation (STW).

† On leave from TNO Physics and Electronics Laboratory, P.O. Box 96864, 2509 JG The Hague.

hidden surfaces removed from an arbitrary point of view. As such, the BSP-tree is especially useful when the object changes less frequently than the eye-position. Only when changing the object, the BSP-tree must be rebuilt.

The algorithm to create the BSP-tree from a list of faces representing the boundary of the polyhedron, is described in *Listing 1*, and illustrated in *Figure 1*, both taken from [2].

To display the polyhedron, the BSP-tree is traversed as described in *Listing 2* (taken from [2]), generating an image with the hidden-surfaces removed. Note that back-facing polygons are not displayed. Displaying them would slow down the display process unnecessarily, as they will be obscured by subsequently drawn polygons. Whether a polygon is back-facing is determined using the polygon's normal, which is set to point away from the polyhedron (see section 5).

The BSP-tree is traversed in linear time O(n), where n is the number of polygons in the BSP-tree. In the BSP-tree the vertices are stored using World Coordinates, so before actually displaying a polygon its vertices are transformed to Device Coordinates. The implementation of this transformation pipeline resembles the one proposed in [3].

4 The "Point in Polyhedron"-test.

With the use of the BSP-tree of the polyhedron we can perform an easy "Point in Polyhedron"-test. Consider the display process using the BSP-tree: the traversal of the BSP-tree is simulated with the eye-position equal to the query point. The point lies inside the polyhedron, if and only if the last polygon considered for drawing is a back-facing polygon. This is because there lie no other faces between the last polygon and the eye-position.

As we are only interested in the last polygon, we do not need to traverse the entire tree. We can speed up the "Point in Polyhedron"-test by skipping the branches which can not contain the last polygon. This results in the algorithm described in *Listing 3*. The worst case execution time for the "Point in Polyhedron"-test is O(n), because the BSP-tree need not be balanced.

5 Computing the direction for the normal vectors.

When traversing the BSP-tree for either display or the "Point in Polyhedron"-test, it is assumed that the normal vector of each face points away from the polyhedron. Hence, after creating the BSP-tree, all normal vectors are checked and multiplied by -1 if not pointing in the right direction. For this it is necessary to be able to distinguish between the inside and the outside of the polyhedron. Our solution is to find a face of which the right direction of the normal can easily be determined, and subsequently compute the right direction of its neighbouring faces. This last step can be continued until all faces have been processed.

To determine the direction of the normal vector of a first face, consider the "Point in Polyhedron"-test discussed in section 4. A one-to-one correspondence is introduced, between the normal vector of the last drawn polygon, and the query point. This correspondence can be used to determine whether a point resides within the polyhedron, knowing the face's normal vector, but can also serve to determine the correct direction of the normal vector, knowing whether the point lies inside or outside the polyhderon. Using the point (infinity,0,0), a point known to lie outside the polyhedron, the direction of the normal vector of the face that would be drawn last can be established.

The following correspondence is defined between the direction of the normal vector and an ordering of the vertices of the polygon: looking at the polygon from the side the normal points to, the vertices should be ordered anti-clockwise; see *Figure 2*. Note that in case of concave polygons the vertices may be ordered clockwise locally. This correspondence can be established relatively easy.

Assuming the normal vector of face F1 to be correct, the corresponding ordering of the vertices is DCBA. Thus edge E is directed from C to B (in face F1; see *Figure 3*). In the neighbouring face F2, the direction of E should be opposite (from B to C), if corresponding to a correct direction of the normal vector for face F2. Thus, the normal vector of F2 can be computed using only the direction of the normal for F1, and the set of vertices for both faces. Hence, we have a method to determine the direction of the normal vector of a neighbouring face. By repeating this step for all neighbours (and their neighbours and so on), the normal vectors of all faces of the polyhedron will point in the right direction. Note that if the polyhedron consists of several unconnected parts, the procedure must be repeated for each part.

6 Checking whether a set of polygons represent a correct polyhedron.

In a first implementation of Hirasp, the BSP-tree was created after the polyhedron was checked. This check, the main bottleneck in the implementation, was caused by the fact that expensive searches had to be performed to find coinciding edges in the unstructured 3D-space.

In an improved implementation this bottleneck is avoided by checking the polyhedron after the creation of the BSP-tree. It allows a more efficient search, because the 3D-space is structured by the BSP-tree. The first tests indicate that this method is about five times faster, and an even greater gain is expected for objects with more faces. A minor disadvantage is that the BSP-tree is also created in the case of an incorrect polyhedron.

References.

[1] W.J.M. Teunissen, J. van den Bos; A Model for Raster Graphics Language Primitives; NATO ASI Series, Vol. F17; Fundamental Algorithms for Computer Graphics; 1985.

[2] Henry Fuchs, Gregory D. Abram, Eric D. Grant; Near Real-Time Shaded Display of Rigid Objects; Computer Graphics, Volume 17, Number 3; July 1983.

[3] Karen Singleton; An Implementation of the GKS-3D/PHIGS Viewing Pipeline; Eurographics 1986.

Listings.

Listing 1: Create the BSP-tree.

```
BSP_tree function MAKE_TREE(polygon_list)
/* This recursive function returns the BSP_tree created from the polygon_list. */
{
        if (polygon_list is empty) return(NULL_TREE);

        root = SELECT(polygon_list);
        back_list = front_list = NULL;
        foreach (polygon in polygon_list and polygon is not the root) {
                case (POSITION(root,polygon)) {
                        FRONT:ADDLIST(polygon,front_list);
                        BACK:       ADDLIST(polygon,back_list);
                        otherwise:{
                                SPLIT_POLYGON(root,polygon,front_part,back_part);
                                ADDLIST(front_part,front_list);
                                ADDLIST(back_part,back_list);
                        }
                }
        }
        return(COMBINE_TREE(MAKE_TREE(front_list),root,MAKE_TREE(back_list)) );
}
```

Listing 2: Traverse the BSP-tree.

```
void function TRAVERSE_TREE(tree)
/* This recursive function traverses an input BSP_tree. */
/* The function DISPLAY handles all transformation and display aspects. */
{
        if (tree is empty) return;

        if (eye is in front of tree.root.polygon) {
                TRAVERSE_TREE(tree.back_descendent);
                DISPLAY(tree.root.polygon);
                TRAVERSE_TREE(tree.front_descendent);
        } else {
                TRAVERSE_TREE(tree.front_descendent);
                /* back-facing polygons are not displayed */
                TRAVERSE_TREE(tree.back_descendent);
        }
}
```

Listing 3: "Point in Polyhedron"-test.

```
boolean function POINT_IN_POLYHEDRON(point,tree)
/* This recursive function returns TRUE if the point lies */
/* inside the polyhedron represented by the input BSP_tree. */
{
        if (point is in front of tree.root.polygon) {
                if (tree.front_descendent is empty) return(FALSE);
                return(POINT_IN_POLYHEDRON(point,tree.front_descendent));
        } else {
                if (tree.back_descendent is empty) return(TRUE);
                return(POINT_IN_POLYHEDRON(point,tree.back_descendent));
        }
}
```

Figures.

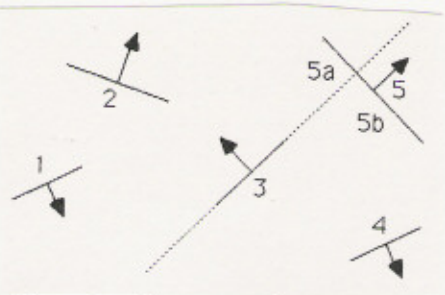

Figure 1.a: Top view of scene.

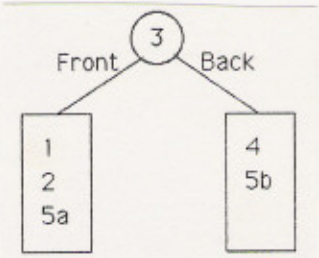

Figure 1.b: After one level of recursion (Polygon 3 chosen as root).

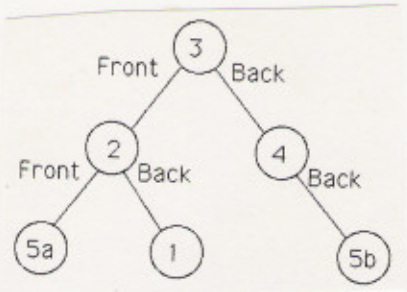

Figure 1.c: A complete tree.

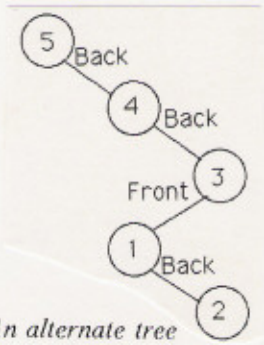

Figure 1.d: An alternate tree with polygon 5 as root.

Figure 1: The building of a BSP-tree (this set of polygons does not represent a polyhedron).

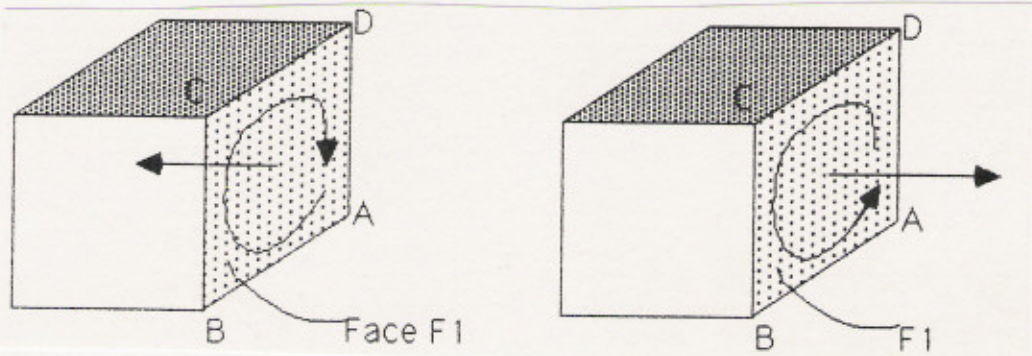

Figure 2: The relation between the direction of the normal vector and the ordering of vertices.

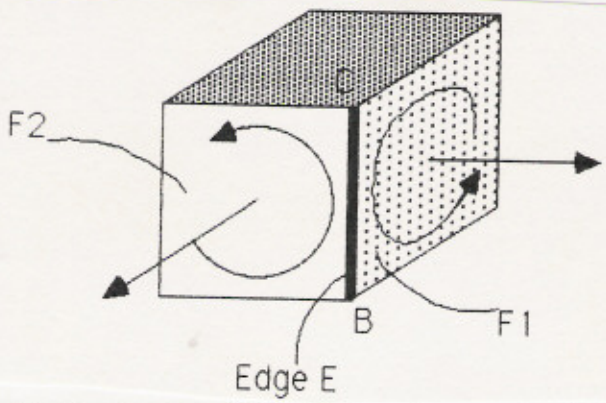

Figure 3: Reversing the direction of E in F2 gives the correct ordering of the vertices.