3-D GEOMETRIC DATABASES USING HIERARCHIES OF INSCRIBING BOXES

W.R. Franklin

Electrical, Computer, and Systems Engineering Dept.
Rensselaer Polytechnic Institute, Troy, N.Y.

ABSTRACT

Hierarchical tree structured databases are an efficient way of representing scenes with elaborate detail of varying scales. Storing a circumscribing box (CIRCUMBOX) around each object is a well known method of testing whether the object intersects or obstructs any other objects. This paper proposes another aid: an inscribing box (INBOX). The inbox is a polyhedron that is completely contained in the object. It should be as large as is easy to determine. If the object is reasonably smooth, but has a lot of surface detail, such as a building, then the inbox will have most of the volume of the object. Each object will now contain: a circumbox, an inbox, possibly subobjects, and possibly primitive elements such as faces. The inbox speeds up visibility tests thus: If the inbox of object A hides the circumbox of object B, then A hides B. If the circumbox of A does not intersect (in projection) the circumbox of B, then A and B are disjoint in projection. Only in the remaining case do we need to consider the subobjects of A and B to determine what is visible. The probability of this happening is determined by the relative volumes of the inbox and circumbox. By combining inboxes and circumboxes, it is possible to calculate the visible surfaces of a hierarchical scene in time linear in the visible complexity of the scene. This is much less than linear in the complexity of the scene (since most of it is hidden), with the difference becoming more pronounced as the scenes get more realistic. The output is calculated exactly to the arithmetic precision of the computer, not just to the accuracy of a pixel.

RÉSUMÉ

Les bases de données à arborescence hiérarchique constituent un moyen efficace de représenter des scènes avec des détails élaborés sur diverses échelles. Le stockage d'une boîte circonscrite (CIRCUMBOX) autour de chaque objet est une méthode d'essai bien connue pour déterminer si l'objet coupe ou obstrue tout autre objet. Le présent document propose un autre moyen: la boîte inscrite (INBOX). La boîte inscrite est un polyèdre qui est complètement contenu dans l'objet. Le polyèdre devra être aussi grand que possible. Si l'objet est relativement uniforme, mais qu'il présente beaucoup de détails superficiels, comme dans le cas d'un édifice, alors la boîte inscrite comprendra la quasi totalité du volume de l'objet. Chaque objet contiendra donc: une boîte circonscrite, une boîte inscrite, possiblement des sous-objets, et possiblement des éléments de base. La boîte inscrite permet d'accélerer les essais de visibilité et par conséquent: si la boîte inscrite d'un objet A cache la boîte circonscrite d'un objet B, alors A cache B. Si la boîte circonscrite de A n'intercepte pas (en projection) la boîte circonscrite de B, alors A et B sont séparés en projection. Il n'y a que dans ce dernier cas qu'il faut considérer les objets de A et B pour déterminer ce qui est visible. La probabilité que cela se produise est déterminée par les volumes relatifs de la boîte inscrite et de la boîte circonscrite. En combinant les boîtes inscrites et les boîtes circonscrites, il est possible de calculer les surfaces visibles d'une scène hiérarchique en temps linéaire dans la complexité visible de la scène. Ceci est loin d'être linéaire dans la complexité de la scène (étant donné que presque tout est caché) avec la différence qui devient plus prononcée è mesure que les scènes deviennent plus réalistes. La sortie est calculée exactement selon la précision arithmétique de l'ordinateur, et non simplement en fonction de la précision d'un pixel.

INTRODUCTION

This paper describes one aspect of a solid modelling system, KEPLER, now being implemented at RPI. It will be useful both for realistic scene portrayal in computer graphics, and object modelling in CAD/CAM.

KEPLER has the following design principles:

1. It is intended to handle large complex scenes with thousands of faces.

2. It builds on recent results in computational geometry that are not widely used in computer graphics since they are so recent, even though they are very powerful.

3. It produces output accurate to the arithmetic precision of the machine. This means that when it calculates the visible surfaces of an object, they are to a precision of one part of 10^7, not just to one pixel. This is important since algorithms that calculate the results to only pixel accuracy require quadruple the time as the linear resolution is doubled. Typical CAD/CAM applications require quite high resolution.

This also means that hierarchical decomposition approximation methods are not adequate since they would require too many levels to achieve that accuracy. A solid modelling system that divides by two in each coordinate at each level would require 23 levels to achieve an accuracy of one part in 10^7. The final level would have at least 10^{15} nodes. Of course, if only 0.001 accuracy is needed, then that method is excellent.

4. KEPLER uses techniques for manipulating large numbers of elements that have been refined in the design of the following algorithms:

a) PRISM [4], which has handled base maps of 13,000 edges (not counting duplications). Other data structures which define each polygon separately and then count all the 3-D edges would count the same scene as having 78,000 edges. A 4,000 edge base map takes about 4 minutes on a PDP KA-10. A sample prism plot is shown in Figure 1. It shows the population in various marketing regions (different urban areas) in the USA.

b) SPHERES [5, 6], which can process 10,000 spheres, overlaid to an average depth of 10, in 383 seconds on a Prime 500. This case is shown in Figure 2. The input scene is so complicated that most of the spheres are completely hidden. SPHERES' algorithm quickly detects and deletes them. The output is exact to single precision accuracy, and independent of the depth complexity of the scene, in contrast to many algorithms whose time grows quadratically with the depth complexity, since they

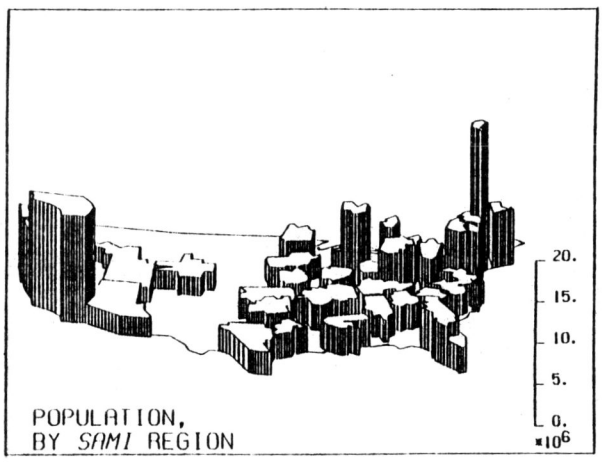

Figure 1: Prism Plot

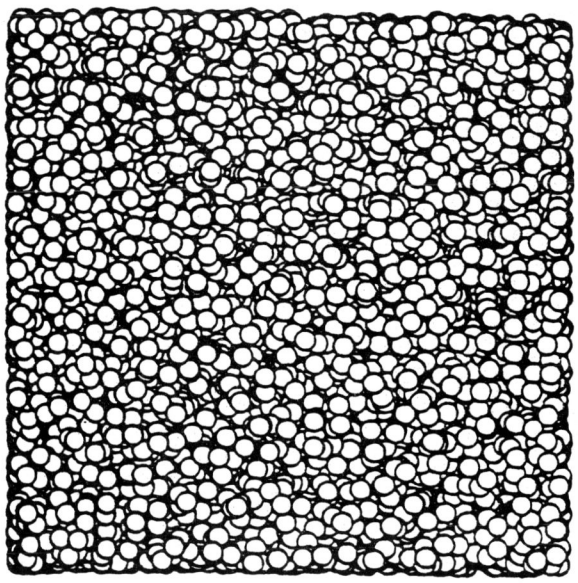

Figure 2: Hidden Spheres Plot

compare pairwise all objects that cross each scan line. The algorithm has no essential dependence on spheres; they just happen to be easy to work with. Varol Akman is now implementing it for general polyhedra.

c) Intersecting edge detection [8], which when given 50,000 random edges, each 0.01 of the screen size long, can find all intersections in 364 seconds on a Prime 500. Figure 3 shows a scene with 1000 edges where the program drew a cross at every intersection that it detected.

The algorithm works by calculating a grid size from the number and length of the edges, and then overlaying the grid on the scene. For each edge, the grid cells it passes through are determined. This information is sorted by cell to produce a list of the edges passing through each cell. For each cell, the edges that pass through it are combined pair by pair to see which intersect. Since the grid size was chosen to keep the average number of edges per cell constant regardless of the input, the execution time is linear in the number of edges plus the number of intersections. In realistic scenes, as the number of edges increases, their average length decreases since details are being represented more accurately. Thus the number of intersections in practice rises much slower than quadratically in the number of edges, so this algorithm is much faster than the simple pairwise comparison of all edges.

d) HALO [8], being implemented by Varol Akman, which can calculate the haloed edge effect for a 9408 edge scene in 5 minutes on a Prime 750. Figure 4 shows 15% of this plot. The entire plot is too complex to reproduce here. What makes HALO efficient is its use of the edge intersection algorithm described above to find which edges pass in front of which other edges.

5. KEPLER is a surface modelling system; that is, the surfaces are explicitly stored, and can be shaded and textured easily. This also makes operations such as intersection simpler. Given the surfaces, solid functions such as volume and moments of inertia can also be quickly calculated. In contrast, with a volume modelling system, it can be difficult to add texture to the surfaces, or even to determine which parts of the surfaces are on the outside of the object.

6. KEPLER currently works with only flat-faced polyhedra. This means that there are fewer details to consider in implementing the efficient algorithms. Once flat objects can be handled fast, and given the routines to intersect and combine curved faces, which must be developed anyway, it is easy to add modules to handle curved surfaces.

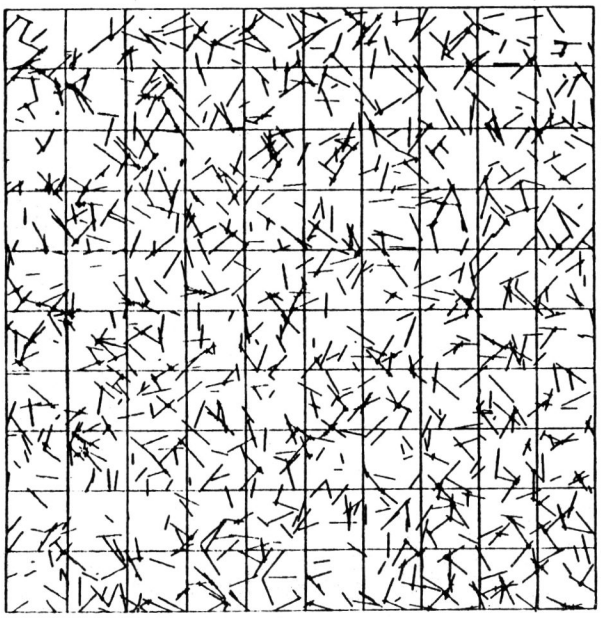

Figure 3: Detecting Edge Intersections

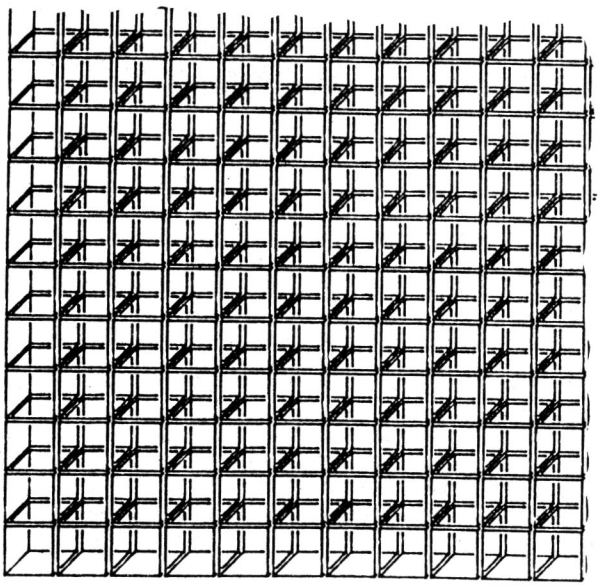

Figure 4: Haloed Line Plot

7. KEPLER is being implemented in Ratfor and Ratmac [9, 10] on a Prime 750. These are preprocessors that accept block structured code with macros and emit Fortran. This is a necessity since Fortran is the only language available. Ratfor is a much better implementation tool than Fortran, though of course much poorer than Lisp. The core of the system, implemented by Bruce Giese, is up and working. It already has over two dozen commands, a database of all the semiregular polyhedra, and extensive documentation.

Thus KEPLER aims to use these techniques to perform operations such as polyhedron intersection, union [7], and hidden surface calculation on big objects quickly. The polyhedron combination algorithms are being implemented by Bruce Giese. They operate by superimposing a grid on the scene, and using it to find all intersections of faces of one polyhedron with the faces of the other. The cut-lines resulting from this are sorted and combined to produce a list of facets with tags. For the desired boolean combination (intersection, union, either difference, or exclusive-or), the facets with the appropriate tags are selected. Thus all the boolean combinations are found at the same time at no increased cost. This algorithm uses no complicated data structures and does not perform complex traversals around the polyhedra. It merely makes successive linear passes through sets of data, calculating new sets which are sorted, combined, and passed through again. Thus it is both efficient and easy to program.

However, for very complex scenes, even techniques such as these may be too slow to calculate the visible surfaces since they require examining each component (edge, face, etc.) of the object at least once, if only to reject it from further consideration.

This paper describes a hierarchical technique that can consider edges and faces in batches. Now hierarchical techniques where each object is composed of subobjects have been described before in many places, such as Clark [2] and Eastman [3]. Enclosing each subobject with a circumscribing box (circumbox) is obvious; then if two objects' circumboxes don't intersect, the objects themselves certainly don't.

However, circumscribing boxes are not adequate for hidden surface calculations. Consider the case where object A is completely hidden by object B. Since A's circumbox is behind B's circumbox, we know that A is probably (not certainly) at least partly hidden. However, to show that A is completely hidden, we must test each face of A against B to show that it is hidden. It would be sufficient if A's circumbox were hidden, since then we would not need to consider any individual faces of A. Even if we couldn't show anything about A's circumbox, we might show something about the circumboxes of A's components. Nevertheless, we must compare these abstractions of A against all the faces of B. We cannot just test A against B's circumbox because the fact that B's circumbox hides something does not mean that B does.

This paper introduces a means of abstracting B, an inscribing box, and presents some algorithms for manipulating it. An inscribing box (inbox) is a rectangular solid that is completely inside the object. Thus if B's inbox hides something, B must hide it. Object B, together with its two boxes, partitions 3-space into 4 regions:

1. Outside B's circumbox,
2. Inside B's inbox,
3. Outside B but inside its circumbox, and
4. Inside B but outside its inbox.

Figure 5 shows the four regions for a simple 2-D object. Thus if we are testing a point P for inclusion in B (which is a necessary step in determining whether object A is hidden), cases 1 and 4 above are easy to determine. We only need to look at B's faces and subcomponents in cases 2 and 3. If B is smooth but has a lot of surface detail, the inbox will be almost as big as the circumbox, so cases 2 and 3 will be infrequent. Of course, other circumscribing and inscribing approximating shapes are possible, and may be appropriate in certain circumstances. Boxes are merely the simplest to manipulate.

Instead of vertices and faces, an object may have subobjects. Each subobject may have either vertices and faces, or subobjects of its own. The previous use of this tree structure, as in Eastman's GLIDE, has been to use multiple copies of a given object, but with certain parameters, such as location, orientation, and size different in each instantiation. This paper uses circumscribing boxes, and adds the inscribing box, to aid the visibility calculations. It is obvious that the well-established features can be added, so it will not be mentioned again.

If desired, this tree data structure may be flattened into a simple object with faces and vertices but no subobjects. The steps to flatten object A are:

1. Flatten any subobject of A that itself has subobjects.

2. Pass through the subobjects of A, and uniquely number the vertices throughout A.

3. Append all the vertex lists and combine all the face sets.

4. Sort the vertex list to find duplications, and combine and renumber them. Update the face set.

This hierarchical structure is explicit, unlike PADL's implicit structure [11, 12]. The implicit structure makes it simpler to build up complicated objects from a small set of primitives, but makes it difficult to determine even such simple operations as the volume or surface area of an object.

In order to efficiently manage the hierarchical scheme of subcomponents and boxes, the technique of surface texture, as described in Atherton and Weiler [1, 13] is used. Texturing involves drawing 2-D points, lines, and polygons on the surfaces of faces.

Texture has at least two uses:

1. It can record surface markings, such as labels. This use is orthogonal to the hidden surface calculations, and so will not be mentioned again.

2. It can record system generated information, such as the visible parts of a face. This capability is necessary in the following algorithms.

It is possible for a subobject to have an ideograph, or a simpler representation that replaces it when the object after scaling is too small. For example, an object which is a tree with every leaf defined might have an ideograph which is just a triangle on a stick. The use of ideographs would require assorted minor changes throughout the following algorithms.

DEFINITIONS AND DATA STRUCTURE

Box: A rectangular solid whose sides are parallel to the axes. A box is defined by six numbers: XL, YL, ZL, XH, YH, ZH.

Circumscribing box (of an object): A smallest box that completely contains the object.

Inscribing box: A largest box that is completely contained by an object. An object may have many inscribing boxes; they may even be disjoint. If the object has no interior points, then every inscribing box is a point.

Object: A component of the scene under consideration, such as a building or a vehicle, that we are manipulating as a unit. An object contains the following components:

1. CBOX, a circumscribing box.

2. INBOX, an inscribing box.

3. {SUBi}, a possibly empty set of subobjects.

4. (VERTi), a possibly empty ordered list of points defined by their Cartesian coordinates. Vertices may not be duplicated; that is, no two vertices may have the same coordinates. However, a vertex of one subobject may have the same coordinates as a vertex of another subobject of the same object.

5. {FACEi}, a possibly empty set of faces. Each face is a planar polygon. No two faces of an object, no matter how many levels deep, may intersect, though they may abut on an edge. Thus when two objects that may overlap are combined, a cleanup routine must be run. Each face has the following components:

   a) The ordered list of the vertices from (VERTi) that are its vertices. Thus if there are faces, there must be vertices.

   b) A possibly empty set of texture. The texture is described below.

If an object has subobjects, then it cannot also have vertices or faces. This implies no loss of generality, since if it had vertices and faces, they could be combined into another subobject of that object.

The components of an object, A, can be accessed by the functions CBOX(A), INBOX(A), NSUB(A), SUB(A,i), NVERT(A), VERT(A,i),

NFACE(A), NVF(A,i), FACE(A,i), and VF(A,i,j). NSUB, NVERT, NFACE, and NVF return the number of subobjects, vertices, faces, and vertices in the i-th face, respectively. FACE(A,i) is the i-th face as a unit, and VF(A,i,j) is the j-th vertex of the i-th face.

The texture of a face contains:

1. A local 2-D coordinate system,
2. A set of points defined in it,
3. A set of lines with endpoints from that set, and
4. A set of polygons with vertices from the point set.

It is possible for the texture to extend outside the face itself, so it need not be clipped when the face is clipped. However, it must be clipped before it is displayed.

The coordinate system is defined by an origin, a unit x-vector, and a parity. The origin is that point of the face plane closest to the 3-D origin. This point is always unambiguously defined. The unit x-vector in the texture coordinate system is in the plane of the face and is defined to be a vector which in 3-space has no z component. If the face is horizontal, the unit x-vector is the (1,0,0) in 3-space. This fixes the unit y-vector to be one of two possibilities. The parity is chosen to make the z component (in 3-space) of the unit y-vector to be positive, unless the face is horizontal, in which case the unit y-vector is (0,1,0).

The above definition makes the face coordinate system independent of the actual face vertices, but dependent on the face plane. An alternative system could be defined that would make the local system dependent on the face vertices but independent of the plane equation. This could have the unit x-vector parallel to the vector from the first to the second vertex, and the unit y-vector at right angles to this such that their cross product was in the same direction as the vector face area. The former method means that local coordinates remain fixed as the face is clipped, while the latter means that they stay the same as the face is rotated.

When the local coordinate system for each face is established, the transformation matrices between the local system and global 3-space are calculated. Note that this transformation does not always change continuously as the orientation of the face changes. This is inherent in 3-D spherical geometry.

ALGORITHMS

This section presents some algorithms that operate on the hierarchical data structure described above. They are chosen to illustrate operations that are much more efficient when an inbox is used. In some of them, the order of certain tests can be interchanged without affecting the result. The order that is used has the least expected execution time for many objects, since it tries tests that are most likely to terminate the algorithm first. Certain low level operations, such as determining whether a point is inside a box are assumed to be available.

A. INCLUDE (A,P)

Input: An object, A, and a point, P.

Output: One of {IN, OUT}, depending on whether P is contained in A.

Algorithm:

1. If P ε INBOX(A), then return IN.

2. Else, if P $\notin$ CBOX(A), then return OUT.

3. Else, if NSUB(A)=0, then apply a standard point inclusion in polyhedron test and return IN or OUT. This takes time linear in the number of faces if no preprocessing is done. If many points are to be tested against a given polyhedron, it can be projected onto a 2-D grid in such a way that each point can be tested in time proportional to the depth complexity of the polyhedron, which is usually very small.

4. Else, if INCLUDE(SUB(A,i),P) = IN for any SUB(A,i), then return IN.

5. Else, return OUT.

B. VISP (A,P)

Input: An object, A, and a point, P.

Output: One of {VISIBLE, HIDDEN}, depending on whether P is visible or is hidden by A when observed from a standard viewpoint, $(0,0,\infty)$.

Algorithm:

1. If P is not hidden by CBOX(A), then return VISIBLE.

2. Else, if P is hidden by INBOX(A), then return HIDDEN.

3. Else, if P is hidden by any face of A, then return HIDDEN.

P is hidden by a face if two conditions are true: P is behind the infinite plane of the face, and P projected onto the X-Y plane is inside the face projected onto the X-Y plane. Testing the first condition is fast if the face equations have been precalculated since which side of aX+bY+cZ+d=0 (x,y,z) is on, depends on whether ax+by+cz+d is negative or positive. Projecting P and the face onto the X-Y plane consists of ignoring the Z-coordinates. Any face that projects to a line is edge-on to the line of sight, and cannot hide P.

4. Else, if VISP(SUB(A,i),P) = HIDDEN for any SUB(A,i), then return HIDDEN.

5. Else, return VISIBLE.

C. VISOBJ (A,B)

Input: Two objects, A and B.

Output: One of {VISIBLE, HIDDEN, MIXED}, depending on whether A does not hide B at all, A completely hides B, or A partly hides B.

Algorithm

1. If NSUB(B) > 0, then do:

   a) If VISOBJ(A,SUB(B,i)) = VISIBLE for all SUB(B,i), then return VISIBLE.

   b) Else, if VISOBJ(A,SUB(B,i)) = HIDDEN for all SUB(B,i), then return HIDDEN.

   c) Else, return MIXED.

2. (B has no subobjects.) Else do:

   a) If VISFAC(A,FACE(B,i)) = VISIBLE for all FACE(B,i), then return VISIBLE.

   b) Else, if VISFAC(A,FACE(B,i)) = HIDDEN for all FACE(B,i), then return HIDDEN.

   c) Else, return MIXED.

D. VISFAC (A,F)

Input: An object, A, and a face, F, of another object.

Output: One of {VISIBLE, HIDDEN, MIXED}, depending on whether A does not hide F at all, A completely hides F, or A partially hides F.

Algorithm

(This algorithm proceeds by keeping track of how much of F has been covered so far as it processes A.)

1. Initialize a visibility texture polygon on F, VTP(F), that exactly covers F. This is the area of F that is not known to be hidden.

2. Execute VISFAC2(A,F).

3. Return VISIBLE, HIDDEN, or MIXED, respectively as VTP(F) is equal to F, is empty, or is neither.

E. VISFAC2 (A,F)

This is the body of VISFAC.

Input: An object, A, and a face, F, of another object. F has a visibility texture polygon, VTP(F).

Output: There is no directly returned value. VISFAC2 operates by the side-effect of modifying VTP(F).

Algorithm:

1. If NFACE(A) > 0, then do:

   a) For each FACE(A,i), calculate the polygon which represents how much of F is visible relative to FACE(A,i), intersect it with VTP(F), and store the result back in VTP(F). If VTP(F) should become empty when executing this loop, return, through any intervening levels of VISFAC2, back to VISFAC.

   b) Return.

2. Else, for each SUB(A,i), do:

   a) If CBOX(SUB(A,i)) does not hide F at all, then proceed to the next iteration of the loop.

b) Else, if INBOX(SUB(A,i)) completely hides F, then set VIP(F) to empty, and return, through any intervening levels of VISFAC2, back to VISFAC.

c) Else, execute VISFAC2(SUB(A,i),F).

SUMMARY

It is possible to manipulate efficiently large objects with complex surface detail and substructures. By using inscribing boxes of subobjects, in addition to circumscribing boxes, this extra detail can be encapsulated so that it does not slow the execution time unduly.

ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under grants no. ENG79-08139 and ISP79-20240.

Varol Akman and Bruce Giese, who are graduate students in Computer and Systems Engineering at RPI, have made excellent implementations of parts of KEPLER.

REFERENCES

[1] P. Atherton, K. Weiler, and D. Greenberg, 'Polygon Shadow Generation', Computer Graphics 12, 3 (August 1978), 275-281.

[2] J.H. Clark, 'Hierarchical Geometric Models for Visible Surface Algorithms', Comm. ACM 19, 10 (Oct. 1976), 547-554.

[3] C. Eastman and M. Henrion, 'GLIDE: A Language for a Design Information System', Computer Graphics 11, 2 (Summer 1977).

[4] W.R. Franklin, 'PRISM - A Prism Plotting Program' in Mapping Software and Cartographic Data Bases, Harvard Library of Computer Mapping, 1979 Collection, edited by Allan H. Schmidt.

[5] W.R. Franklin, 'A Linear Time Exact Hidden Surface Algorithm', Computer Graphics (ACM-SIGGRAPH) 14, 3 (July 1980), 117-123.

[6] W.R. Franklin, 'An Exact Hidden Sphere Algorithm That Operates in Linear Time', Computer Graphics and Image Processing 14, 4 (April 1981), 364ff.

[7] W.R. Franklin, Efficient Polyhedron Intersection and Union, Rensselaer Polytechnic Institute, Image Processing Lab, IPL-TR-81-001 (Jan. 1981).

[8] W.R. Franklin, Efficiently Computing the Haloed Line Effect for Hidden Line Elimination, Rensselaer Polytechnic Institute, Image Processing Lab, IPL-TR-81-004 (Dec. 1980).

[9] B.W. Kernighan and P.J. Plauger, Software Tools, Addison-Wesley (1976).

[10] R.J. Munn and J.M. Stewart, RATMAC Primer, University of Maryland Division of Agricultural and Life Sciences, TR-804 (August 1979).

[11] R.B. Tilove, 'Set Membership Classification: A Unified Approach to Geometric Intersection Problems', IEEE T. Comp. C-29, 10 (Oct. 1980), 874-883.

[12] H. Voelcker, et al., 'The PADL-1.0/2 System for Defining and Displaying Solid Objects', Computer Graphics 12, 3 (August 1978), 257-263.

[13] K. Weiler and P. Atherton, 'Hidden Surface Removal Using Polygon Area Sorting', Computer Graphics 11, 2 (Summer 1977), 214-222.

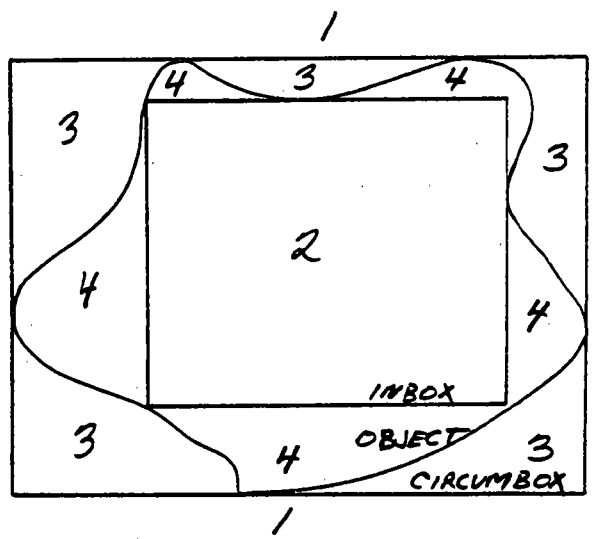

Figure 5: Circumscribing and Inscribing Boxes