A SIMPLE BUT SYSTEMATIC CSG SYSTEM

Tosiyasu L. Kunii
University of Tokyo, Japan

Geoff Wyvill
University of Otago, New Zealand

ABSTRACT

A simple system is described for computer aided design by constructive solid geometry (CSG). The system allows the design of engineering components by combining 'basic components' which represent shapes produced by standard machining operations.

The system has four significant original features:

1. The primitive components are described in an object oriented fashion, data plus procedures.

2. A new kind of octree structure is used to render various displays from descriptions.

3. Certain objects can be directly associated with components of tool paths. For example, a cylindrical object might represent a drill moving along its length or a prism might represent the shape of material cut by a milling tool sweeping horizontally. An object built from these basic objects can, in principle, be cut using combinations of their associated tool-paths.

4. The system consists of four conceptual modules with well-defined interfaces. One module, for example, is the set of primitive objects and their associated procedures. Another is the octree generator. Because of this design technique, it is easy to modify or even replace modules. This meta-structure provides us with a general, if informal method of describing CSG systems.

KEYWORDS: CAD/CAM, Geometric modelling, CSG, Octree, Ray tracing.

INTRODUCTION

In the last few years the method of constructive solid geometry (CSG) has become increasingly popular as an alternative to surface-based models for Computer Aided Design [1, 2, 3, 11, 13, 14, 20, 22]. In a CSG system, objects are represented as collections of 'primitive objects' connected by set operations on the space they occupy. This leads, naturally, to the representation of objects as a tree structure where the leaves represent primitive objects. The nodes, other than leaves, represent set operations between sub-objects. Since the sub-objects can be different copies of the same object description, we prefer to represent this as a directed acyclic graph (DAG) or node-sharing tree. The edges of this graph carry geometrical transformation information. (Diagram 1.)

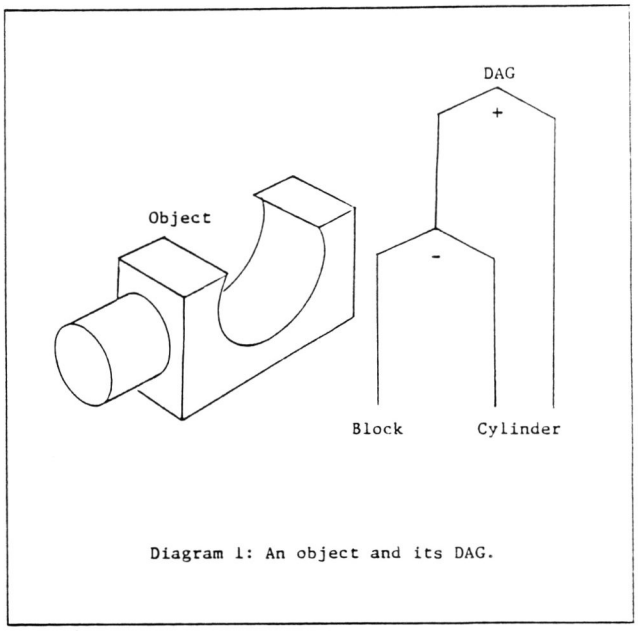


Diagram 1: An object and its DAG.

This structure is also a natural extension of the recursive picture languages PG, Pictures 68 and PDL-2 [9, 23, 24]. The structure can be built from text descriptions, algebraic descriptions or, in principle, interactively with a graphics console. The user interface is a substantial separate problem which we intend to address in a subsequent paper.

In order to extract useful information from the DAG, a rendering process is needed. Usually, this implies building a secondary structure

which can be processed by conventional display algorithms. For this structure, surface models have been used [2, 3, 13 and others] and more recently octree and other solid grid representations [8, 10, 25]. We use a modified octree structure. It provides a spatial ordering to reduce the complexity of a scene (e.g. for ray tracing) while retaining almost all of the information from the primitive CSG objects.

Each primitive object is represented as a collection of procedures which describe its properties. Some of these procedures are called by the rendering algorithm as it processes the intermediate (octree) structure. Others are called as the octree is being built. This means that new primitives can be added in a systematic way, by writing procedures which describe them.

Similarly, new rendering algorithms can be added, for example to generate n.c. machine tool paths. Depending on its purpose, an algorithm can extract information from the DAG, the intermediate structure or both.

The system consists of four parts.

1. The primitives and their procedures.

2. The CSG structure and user interface.

3. The intermediate structure and its creation algorithm.

4. The renderers

These communicate through well-defined interfaces. In principle, any part can be replaced by a functional equivalent, although in the pilot version, the renderers include some knowledge of the intermediate structure and a radical change in the CSG structure would require changes in the 'creation procedure'.

THE DAG STRUCTURE

The language PDL-2 [24] is for describing pictures. The line:

DEFINE stool leg leg AT 1,0 top AT 0,1 TURNED 90

defines a (2-D) object called "stool" which consists of two copies of the object called "leg" and a copy of the object "top". The phrases "AT 1,0" and "AT 0,1 TURNED 90" convey information about the position and orientation of the copies of "leg" and "top". The representation of this in DAG form is shown in diagram 2.

A definition is a list of "instances" of subpictures and each instance includes a pointer to a definition.

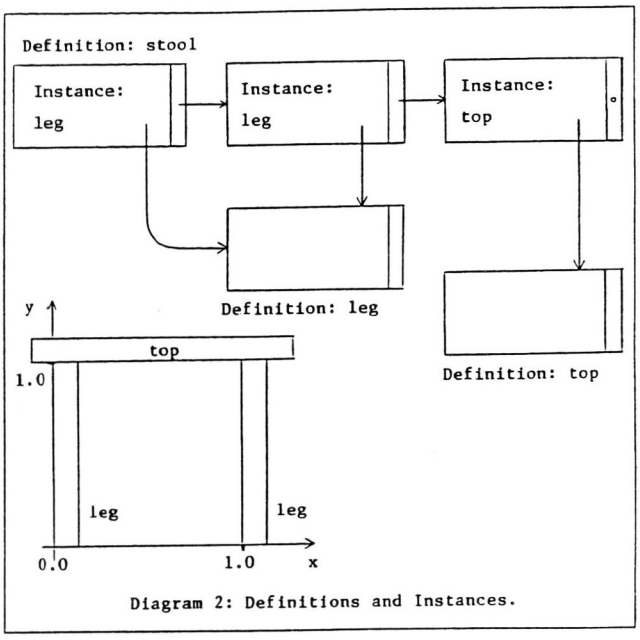


Diagram 2: Definitions and Instances.

Similarly in our DAG structure a list of elements constitutes an object's definition and each element contains a pointer to a definition. Our DAG elements have five fields:

Mode: PLUS or MINUS

Trans: A pair of matrices which describe the position of this instance in the current definition.

This: A pointer to another object's definition. (This instance.)

Next: A pointer to the next element of this definition.

Props: A pointer to another structure containing properties of this instance.

The mode PLUS indicates that this object is to be added in the current definition. Mode MINUS indicates that the object represents a volume cut away.

The matrices are simple matrices of linear transformation. For example:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} ax+by+cz+d \\ ex+fy+gx+h \\ ix+jy+kz+1 \\ 1 \end{pmatrix}$$

This matrix transforms the point <x,y,z> into some other coordinate system. The elements d,h,l describe a translation of <x,y,z> and the

other elements describe rotation and magnification. Because matrix multiplication is associative, we can accumulate a number of matrices, M1*M2* ... Mn and observe that for a position p=<x,y,z>:

M1*M2*... Mn*p = M1*(M2*(M3... (Mn*p)))

We use this in traversing the DAG to find the final position of each primitive element in the world space. (See [23, 24]).

The properties are other information that can be associated with an object. In our pilot system, the only properties are surface colour and reflectivity, but other information such as density, elasticity or material cost could be represented according to the needs of applications. The pointer "This" can also point to a 'primitive object' which is a collection of procedures. In this case, we say the DAG element is 'primitive'.

THE INTERMEDIATE STRUCTURE

An octree model [8, 10, 25 and others] divides a cubic region of space recursively into eight sub-cubes at each level of a tree structure. Each leaf of the tree represents an undivided cube which is either 'full' or 'empty'. 'Full' elements can have other associated information e.g: colour. A modified structure has also been described, in which the leaves of the tree contain a strictly limited set of surface elements [4, 25]. This has the advantage of compactness. Also less information is lost in transforming to this model.

Our intermediate structure is a modified octree in which space is divided until each leaf:

1. is Full,
2. is Empty,
3. contains boundaries between empty space and one primitive object,
4. contains boundaries between full space and one primitive object (In this case the object is being subtracted from full space), or
5. represents a volume of space less than the limit of system resolution.

Nodes of the tree, other than leaves, represent cubes which are further divided into eight sub-cubes recursively until a leaf node is reached. Such cubes and nodes are called 'partial'. Note that the leaf nodes in (5) above, would also be partial and have 'children' but for the limit on resolution. We refer to these cells as "nasty" as there is no easy way to extract information from them. Because they occupy only a small proportion of the total space, this is acceptable.

See also, that this structure differs from [4, 25] in that the leaf nodes refer to a primitive solid object and not to part of a plane, edge or vertex.

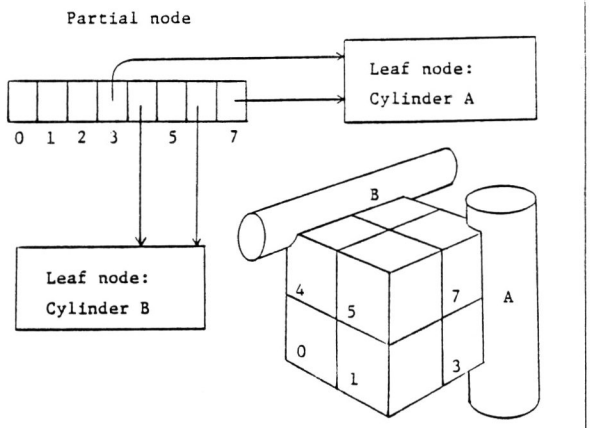


Diagram 3. Modified Octree Structure.

Note the sharing of leaf nodes.

THE CREATION ALGORITHM

Creation of the octree is accomplished in two phases. First the DAG structure is traversed top-down using a current transformation matrix, 'forward' which describes the position and orientation of the current object in the world space. There is also a 'return' matrix which is the inverse of the 'forward' one.

As each DAG element is encountered, this current matrix is pre-multiplied by the element's own matrix to obtain the transformation for the sub-components of the DAG. At the same time a new 'return' matrix is computed by pre-multiplying the 'return' matrix of the DAG element and the current 'return' matrix. This procedure is repeated, recursively on the DAG until a primitive element is encountered. At the primitive element, therefore, we have a matrix which describes the position of the primitive object in world space, and also a matrix for the inverse transformation.

When a primitive element is encountered, we create an elementary octree structure consisting of a single leaf. The matrices are copied into

the octree leaf node and the mode is set to PLUS. These elementary structures are merged into a single tree in phase two.

The traversal algorithm can be summarised thus:

1. If DAG is empty, return the empty octree.
2. Traverse DAG.next recursively to obtain an octree for the background to which this element will be added or subtracted.
3. Calculate new matrices for DAG.
4. If DAG is primitive create an octree leaf element called foreground.
5. If DAG is not primitive, traverse DAG.this recursively using the new matrices to return an octree called foreground.
6. If the mode of DAG is PLUS, add the foreground to the background and return the result.
7. If the mode of DAG is MINUS, subtract the foreground from the background and return the result.

The addition and subtraction phase is thus separate from the traversal only in a conceptual sense. Add and subtract are called as subprocedures of the traversal algorithm.

To add two octrees a,b we proceed as follows:

1. If a is empty, result is b, otherwise:
2. If b is empty, result is a, otherwise:
3. If a is full, then the objects interfere. This is not allowed.
4. If b is full, the objects interfere, otherwise:
5. If we have reached the limit of resolution, create a nasty (type 5) cell, otherwise:
6. Subdivide a and b. If a or b is a partial node, this just means we access its child nodes. If a or b is a leaf node, we create child nodes for it.
7. Add the eight sub-elements of a,b recursively.

Notice that we do not permit conventional set union. It is a feature of the system design that we imitate nature. We do not permit two objects to occupy the same space. This check for interference also is the basis for our approach to tool path generation.

To subtract two octrees a,b we proceed as follows:

1. If a is empty, result is empty, otherwise:
2. If b is empty, result is a, otherwise:
3. If b is full, result is empty, otherwise:
4. If a is full and b is a leaf node, return the inverse of b, otherwise:
5. If we have reached the limit of resolution, create a nasty (type 5) cell, otherwise:
6. Subdivide a and b as for addition.
7. Subtract the eight sub-elements of a,b recursively.

When we subdivide a primitive octree, we construct the co-ordinates of the boundaries of the eight sub-cubes. Now using the 'return' matrix of the leaf element, we transform the co-ordinates of the voxel boundary back into the space belonging to the primitive. These transformed points are passed to the appropriate primitive routine which returns one of three values:

IN indicates that the voxel boundary is completely contained. This means that the voxel is full. (Empty in subtractive mode.)

OUT indicates that the voxel boundary is completely outside the space occupied by the primitive; the reverse of IN.

BORDER means that the child voxel still contains part of the primitive boundary.

Thus some of the child nodes will be full or empty and this enables the recursive calls of add and subtract to do their job. Others will require further subdivision.

The only information required of the primitives to build the intermediate structure, is defined through one uniform procedure interface. This is an example of the value of our meta-structure.

THE RAY TRACER

To illustrate the octree algorithms, we have written a simple ray tracer which operates on an octree structure. Since each leaf voxel refers to only one primitive, the ray tracer does no searching among objects. Instead, each ray is

followed through the structure until it encounters a non-empty voxel. At this point, a primitive function is called to deal with the intersection.

The rest of the ray tracer's function requires knowledge of surface properties which are described in the DAG and octree structure and are independent of the primitives. This information comes fom the octree leaf.

To achieve a reasonable speed, it is important that we skip over empty voxels fast. Our 'next voxel' algorithm is a refinement of Glassner's [7].

THE NEXT VOXEL ALGORITHM

Like Glassner [7], we find a point which is guaranteed to be in the next voxel. But we avoid most of the computation of solving plane intersections as follows. (Uppercase is used for vectors so P[z] is the z component of P.)

Consider a ray from P to Q within a voxel v. Let D=Q-P and note that D does not change during the traversal of many voxels.

Now find R such that R[i] (i=x,y,z) is the distance from P to the exit point of v in direction i. R[i] will have the same sign as D[i].

Now find t such that t is the minimum value of:

abs(R[x]/D[x]), abs(R[y]/D[y]) and abs(R[z]/D[z])

The point P[x]+tD[x], P[y]+tD[y], P[z]+tD[z] is guaranteed to lie on the voxel boundary. This, of course, assumes that we can perform the divisions in an exact mathematical way with no rounding error.

To avoid floating point calculation and its attendant uncertainties we have to rearrange the calculation slightly. For a voxel bounded by L,H (bottom southwest and top northeast corners) we consider P to be inside v if L[i]<=P[i]<H[i], (i=x,y,z). If D[i] is positive, then:

R[i]=H[i]-P[i] but if D[i] is negative, let:

R[i]=L[i]-P[i]-1

so R[i] is the minimum movement on axis i which takes us into the next voxel.

The divisions (above) don't work with integers. D[i] is usually greater than R[i]. So instead we find the direction of movement by cross multiplication. For example: if R[x]D[y] < R[y]D[x] it indicates that the ray will intersect the voxel on the x co-ordinate before the y co-ordinate. The new P is thus found from the old as follows:

```
for i:=x to z do
    if D[i]>=0 then R[i]:=H[i]-P0[i]
    else R[i]:=L[i]-P0[i]-1;
k:=x;
for i:=y to z do
    if abs(R[k]*D[i])>abs(R[i]*D[k]) then k:=i;
for i:=x to z do
    if i=k then P[i]:=P0[i]+R[i]
           else P[i]:=P0[i]+(D[i]*R[k])/D[k];
```

This works even when the smallest voxel is only one unit wide. The truncation error in the division is always a fraction of a unit. And in the awkward case where the ray reaches two or three boundaries simultaneously, there is no error.

Since we consider only the exit surfaces of the voxel for the ray, we repeat this calculation for each voxel always using the original value of P, P0. Thus the truncation errors from the divisions do not accumulate and each P is guaranteed to be in the correct voxel. Notice that R[i] can never be zero, so that D[k] can only be zero in the trivial case where D[x], D[y], D[z] are all zero.

This algorithm uses only eight integer multiplications and two divisions per voxel.

Once we have a point guaranteed to be in the correct voxel, finding the voxel is straightforward (see Glassner[7]).

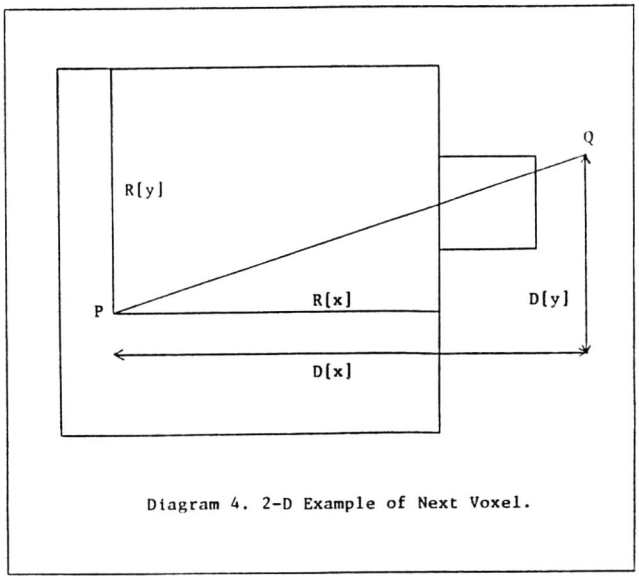


Diagram 4. 2-D Example of Next Voxel.

THE RAY TRACING PRIMITIVE ROUTINES

Having found a non-empty voxel, the ray tracer must analyse the ray impact. This is done by a procedure from the appropriate primitive. First the intersection points of the ray with the voxel boundary are found. These points are then transformed back into the primitive's space using the inverse matrix associated with the octree leaf. The primitive procedure, therefore, is presented with the end points of a short ray. It returns TRUE or FALSE depending on whether the ray intersects the ideal object between its end points. When intersection occurs, it also returns the intersection point, and two others from which the ray tracer finds the surface normal. We don't allow the primitive routine to find the surface normal, because that would limit us to transformations which preserve angles. Our DAG building routines include stretching operations so that we can make ellipsoids from spheres and cylinders.

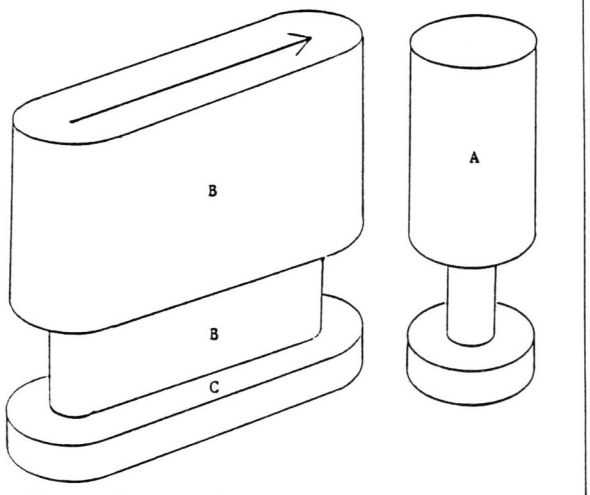


Diagram 5. An idealised milling tool, A and its associated object. B is added. C is subtracted. The arrow shows the tool path.

THE SYSTEM PRIMITIVES

An important feature of our system is the logical separation of procedures which describe primitives from the rest. These procedures provide us with a very flexible way to describe the nature of the primitive objects.

To date, we have implemented only a plane halfspace and a cylinder as primitives. Because the only functions of our system are octree building and ray tracing, each of these primitives has only two associated procedures: "in" and "intersect".

"In" takes as arguments, eight points which are the transformed corners of a voxel. It tells us whether the voxel is completely inside the primitive, completely outside or neither. "Intersect" takes two points as arguments and finds intersections as described above.

Other routines which could be associated with primitives would perform functions appropriate to the needs of a system's applications. For example:

1. A volume procedure returning the proportion of a voxel's volume occupied.

2. A sketch procedure which generates line segments for sketching that part of a primitive lying inside a voxel.

3. A tooling procedure which creates elements for tool path generation.

TOOL PATH GENERATION

One of the objectives in our design is to facilitate the generation of tool paths for the automatic machining of objects described by the system. Yamaguchi, Kunii et al. [25] have described the generation of a simple tool path from an octree. Our system is designed to generate tool paths from the description in the DAG. The octree is used only as a check for interference.

Consider the operation of a drill. The volume cut out is described by a (subtracted) cylinder. The volume swept out by the chuck holding the drill can similarly be described by a cylinder, possibly with a shaped front end. We can describe the drilling action as the subtraction followed by adding the cylinder representing the chuck. If the chuck movement interferes with any part of the work piece, this will be detected during the octree construction.

Similarly a milling operation can be described by combining the shapes of diagram 5.

The use of a tool is described thus:

1. Subtract tool volume.

2. Add 'head' volume.

3. Subtract 'head' volume.

This should produce the same object as step 1 alone. For tool path generation, we start by describing the work piece. We then describe the finished shape by subtracting from the work piece, a sequence of objects each having an associated tool path component. Then we generate

an octree and if no interference is detected, we have described an object which can be created by using the specified cutting operations in sequence. Finally, we can use the ray tracer to display the object. If it appears correct, then we have a sequence of machine operations to produce it.

In complicated cases, we can combine objects which represent a sequence of cutting operations. The tool path sequence is then determined by a top down traversal of the DAG.

DISCUSSION

The simple octrees of Tanimoto[8] and Meagher[10] have a number of leaves approximately proportional to the surface area of the objects represented. Our octrees have a number of leaves approximately proportional to the sum of the lengths of all the edges of the objects. An edge is, by definition, a line along which two primitive objects meet. It would be interesting to experiment with a system which permitted up to two primitive objects per voxel instead of one. Long edges would then be contained in large voxels and the 'nasty' cells would be confined to places where three or more primitives intersect.

Our algorithms are not particularly fast or space saving. For example, in the ray tracer, we transform each ray into a different space for processing by the primitive. Perhaps it is possible to create a package of data at octree creation which the primitive can use to operate on the ray in its original coordinates. This might take longer, but it need be done only once for each leaf in the octree, rather than many times in the ray tracer.

Although our ray tracer is not fast (50-100 rays per second on a VAX 750), it must be remembered that each intersection is calculated with respect to an ideal primitive object. We do not get the faceted look of cylinders aproximated by swept polygons.

We do not handle colours very well. When we make a hole, for example, by subtracting a blue cylinder from a red cube, the inside of the hole is blue! This is because we have only one kind of 'full' voxel. Arranging proper inheritance of colour or other properties is a substantial problem demanding attention.

The need for sets of points in space to be regular has been pointed out by Tilove [17, 18] and Voelcker and Requicha [14]. In essence a regular set of points is one from which dangling points, lines and planes of no thickness have been removed. Since our system allows for arbitrary set subtraction, it is capable of generating such dangling points and they are represented by unnecessary 'nasty cells' at the limit of resolution. These cells can be eliminated and we call this process regularisation. At the time of writing, we have designed a regularisation algorithm, but it has not been tested. Regularisation of an octree with superfluous cells reduces the storage needed. It may prove desirable to regularise the octrees during creation. We hope to report on this later.

ACKNOWLEDGEMENT

We are grateful to The Software Research Centre of Ricoh Co., Ltd. for financial support in this project.

CONCLUSION

A very simple pilot system for CAD by CSG has been produced to test some new principles and algorithms. Preliminary results suggest that this is a promising approach but more experimentation is needed to assess its performance. A new octree related structure has been designed and algorithms appropriate to its exploitation have been written. This new structure retains in its leaf elements, references to descriptions of the system primitive objects. This avoids the loss of information usually associated with this kind of structure.

REFERENCES

1. Atherton Peter R.
"A Scan-line Hidden Surface Removal Procedure for Constructive Solid Geometry"
Computer Graphics
Vol. 17 No. 3 July 1984 pp 73-82

2. Boyse, J.W. and Gilchrist, J.E.
"GMSolid: Interactive Modelling for Design and Analysis of Solids"
IEEE Computer Graphics and Applications
Vol. 2 No. 2 March 1982 pp 86-97

3. Brown, C.M.
"PADL-2: A Technical Summary"
IEEE Computer Graphics and Applications
Vol. 2 No. 2 March 1982 pp 69-84

4. Carlbom, Ingrid; Chakravarty, Indranil; Vanderschel David
"A Hierarchical Data Structure for Representing the Spatial Decomposition of 3D Objects"
Frontiers in Computer Graphics: Proceedings of Computer Graphics Tokyo '84
Springer-Verlag 1985

5. Dippe, Mark; Swenson, John
"An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis"
Computer Graphics
Vol. 18 No. 3 July 1984 pp 149-158

6. Frieder, Gideon; Gordon, Dan; Reynolds, Anthony R.
"Back-to-front Display of Voxel-Based Objects"
IEEE Computer Graphics and Applications
Vol. 5 No. 1
January 1985 pp 52-60

7. Glassner, Andrew S.
"Space Subdivision for Fast Ray Tracing"
IEEE Computer Graphics and Applications
Vol. 4 No. 10 October 1984 pp 15-22

8. Jackins C.L.; Tanimoto S.L.
"Oct-trees and Their Use in Representing Three Dimensional Objects"
Computer Graphics and Image Processing
Vol. 14 No. 3 November 1980 pp 249-270

9. Liblong B; Hutchison N
"PG A Graphical Editor"
University of Calgary, CPSC project 1982

10. Meagher D.
"Geometric Modeling Using Octree Encoding"
Computer Graphics and Image processing
Vol. 19 1982 pp 129-147

11. Myers, W.
"An Industrial Perspective on Solid Modeling"
IEEE Computer Graphics and Applications
Vol. 2 No. 2 March 1982 pp 86-97

12. Norio Okino, Yukinori Kakazu, Masamichi Morimoto
"Extended Depth-Buffer Algorithms for Hidden Surface Visualization"
IEEE Computer Graphics and Applications
Vol. 4 No. 5 May 1984

13. Requicha, A.A.G. and Voelcker, H.B.
"Geometric Modelling of Mechanical Parts and Processes."
IEEE Computer
December 1977 pp 48-57

14. Requicha, A.A.G. and Voelcker, H.B.
"Solid Modeling: A Historical Summary and Contemporary Assessment"
IEEE Computer Graphics and Applications
Vol. 2 No. 2 March 1982 pp 9-24

15. Roth, S.D.
"Ray Casting for Modelling Solids"
Computer Graphics and Image Processing
Vol. 18 1982 pp 109-144

16. Sequin C.H.; Strauss P.S.
"Unigrafix (three dimensional graphics modelling)"
ACM IEEE 20th Design Automation Proceedings
1983 pp 374-381

17. Tilove, R.B.
"Set Membership Classification: A Unified Approach to Geometric Intersection Problems"
IEEE Trans. Computers
Vol. C-29 No. 10 October 1980 pp 874-833

18. Tilove, R.B.; Requicha, A.A.G.
"Closure of Boolean Operations on Geometric Entities"
Computer Aided Design
Vol. 12 No. 5 September 1980 pp 219-220

19. Tuy, Heang, K.; Tuy, Lee Tan
"Direct 2-D Display of 3-D Objects"
IEEE Computer Graphics and Applications
Vol. 4 No. 10 October 1984 pp 29-33

20. Voelcker, H.B.
"Algorithms and Applications"
Tutorial on Solid Modelling
SIGGRAPH '82 (ACM)

21. Whitted, J. Turner
"An Improved Illumination Model for Shaded Display"
CACM
Vol. 23 No. 6 June 1980 pp 343-349

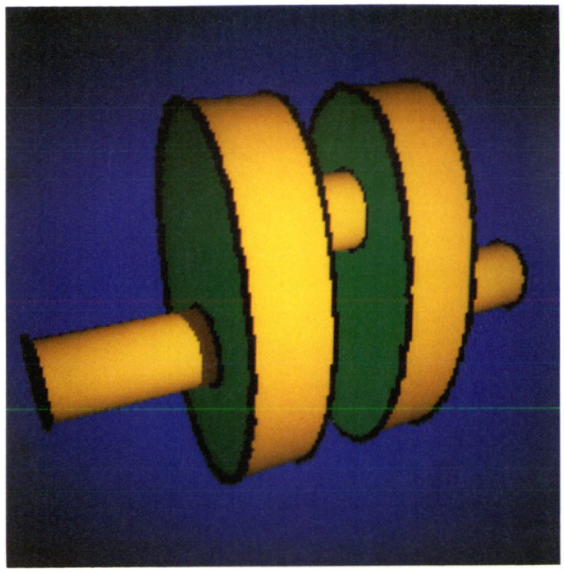
22. Wolfe, R., Fitzgerald, W. and Gracer F.
"Interactive Graphics for Volume Modeling"
Proceedings of the IEEE Eighteenth Design Automation Conference
pp 463-470

23. Wyvill, B.L.M.
Pictures-68 MK1
Software: Practice and Experience
Vol. 7 No. 2 1977 pp 251-261

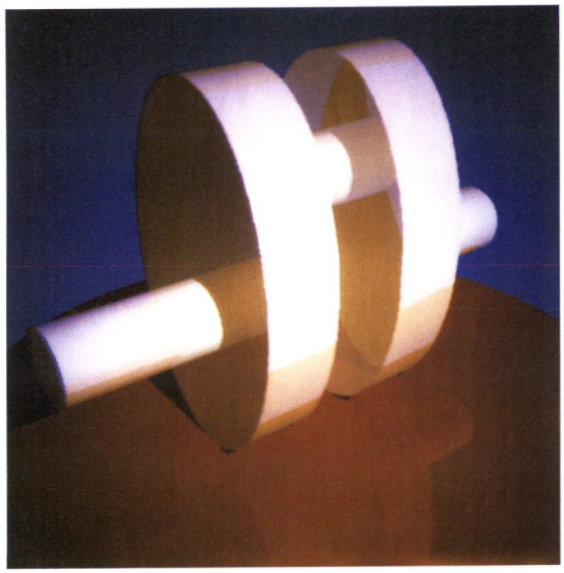
24. Wyvill, G
"Pictorial Description Language 2"
Interactive Systems
Proceedings of The European Computing Conference
Brunel University 1975

25. Yamaguchi, K.; Kunii, T.L.; Fujimura, K.; Toriya, H.
"Octree Related Data Structures and Algorithms"
IEEE Computer Graphics and Applications
Vol. 4 No. 1 January 1984 pp 53-59

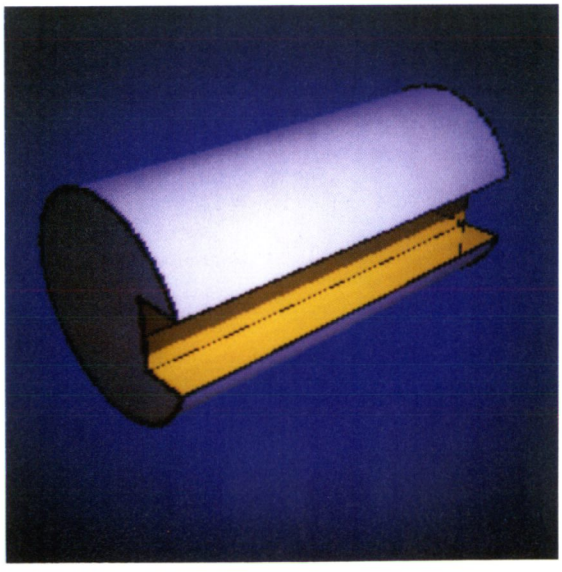
1



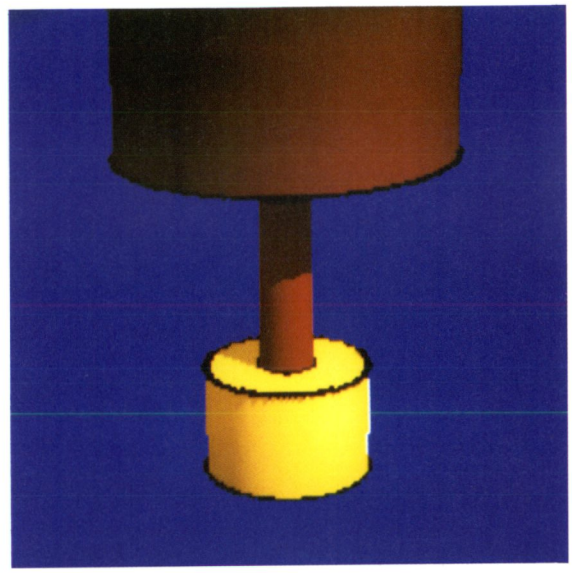
2



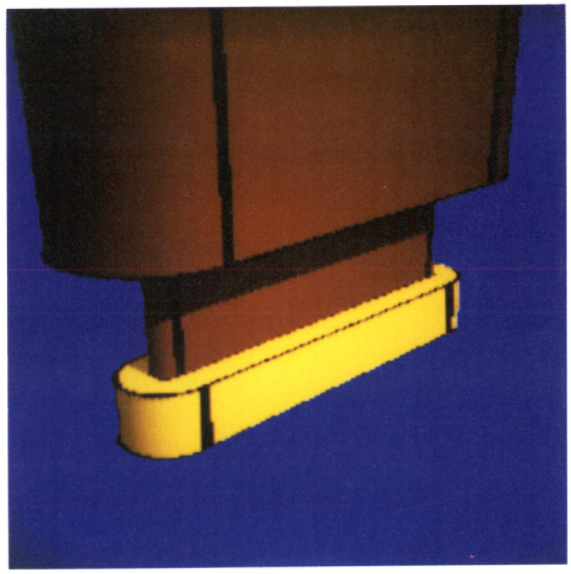
3



4



5



1. A crankshaft shown at low resolution (Octree depth 8). The 'nasty' cells appear black. Note that the webs are elliptical and not circular.
2. The same crankshaft at higher resolution (Octree depth 10). The surface has been made reflective and the 'nasty' cells have been disguised by colouring them the same as an adjacent pixel. The shaft rests on a polished red table and reflections in the various surfaces add a touch of realism.
3. A short shaft in which a keyway has been cut.
4. The idealised Milling tool used to cut the keyway.
5. The toolpath object which represents the cutting of the keyway. The yellow part represents the path of the tool to be subtracted from the shaft. The red part is the shape which is added to test for interference.