An Improved Z-Buffer CSG Rendering Algorithm

Nigel Stewart *
Department of Manufacturing Systems Engineering

Geoff Leach
Department of Computer Science

Sabu John
Department of Manufacturing Systems Engineering

RMIT University, Melbourne, Australia

Abstract

We present an improved z-buffer based CSG rendering algorithm, based on previous techniques using z-buffer parity based surface clipping. We show that while this type of algorithm has been reported as requiring O(n^2), (where n is the number of primitives), an O(kn) (where k is depth complexity) algorithm may be substituted. For cases where k is less than n this translates into a significant performance gain.

CR Categories: I.3.5 [Computing Methodologies]: Computer Graphics—Constructive solid geometry (CSG) I.3.3 [Computing Methodologies]: Computer Graphics—Display Algorithms I.3.1 [Computing Methodologies]: Computer Graphics—Hardware Architecture

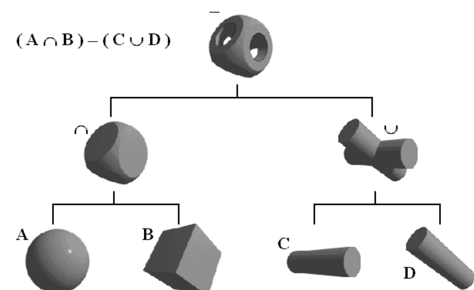


Figure 1: A CSG tree.

1 INTRODUCTION

Constructive Solid Geometry (CSG) is an approach to geometric modeling. CSG arranges boolean operations and primitive objects into a tree. The nodes (or non-terminals) of the tree represent boolean operations, and the leaves (or terminals) represent objects. The boolean operations used in CSG are Union (∪), Intersection (∩) and Difference (-). Affine transformations such as scale, translation and rotation may also be associated with each tree node. Figure 1 illustrates an abstract CSG object specified in terms of boxes, spheres and cylinders.

CSG has significant advantages for some problem domains. One such domain — the main motivation for our work — is Computer Aided Design (CAD) packages for modelling components and devices manufactured using processes, such as milling and grinding, where material is removed from an initial work-piece, typically a block or cylinder. A machine tool produced by this kind of process is shown in Figure 2.

For each milling or grinding operation, the volume swept by the cutting tool can be tessellated into triangles. This boundary representation (b-rep) surface must then be clipped against the work-piece and all other operations in order to display the surfaces that form the final component or device. This is the central problem in solid modelling applications. Broadly, there are two classes of approach: object space and image space. Object space approaches such as analytic intersection, b-rep and spatial enumeration schemes apply clipping operations to objects represented in terms of geometric primitives. Image-space approaches such as ray-casting, scan-line and z-buffer algorithms perform clipping as part of the rendering process, and operate on pixels. Object space approaches are viewer independent, while image-space approaches are not. The cost of object space calculation is typically higher than the cost of one frame of image space calculation. Since image space approaches regenerate the solution each frame, they are better suited to situations where the position, shape and relationship of objects vary over time.

This paper presents an image-space z-buffer CSG rendering algorithm based on the previous work of Goldfeather[5, 6]. We propose re-arranging the algorithm in order to exploit image-space characteristics of the scene. We detail the Goldfeather algorithm, introduce a new variation, and compare performance characteristics.

*Email: nigels@eisa.net.au

2 CSG RENDERING

There are different ways of rendering the image of a CSG tree. Here we give an overview of the main approaches and discuss their advantages and disadvantages. We use the term *clipping* in a generic sense, referring to any process of classifying the surface of one volume with respect to the surface of another volume.

B-rep [12, 13] Each primitive is tessellated into a set of polygons, typically triangles. A clipped b-rep is formed by finding the portions of the surface which satisfy the CSG tree logic. The result of the b-rep clipping operation can be passed to a rasteriser as a set of polygons. The main advantage of this approach is the view independence of the solution.

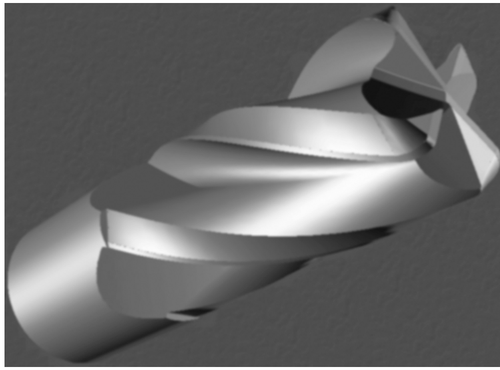


Figure 2: Simulated manufactured object.

Ray Casting Surface clipping is simplified by only considering one line in space at a time. For each pixel, a *ray* is formed which is then intersected with the objects in the CSG tree. The closest visible surface is resolved on the basis of the closest surface satisfying the CSG tree logic. One advantage of ray casting is that the line-object intersection can be implemented for a broad class of objects including analytic and tesselated surfaces. Ray casting is considered to be easy to implement and more robust than the b-rep approach. The disadvantages of ray casting are that the solution is view dependent, each image can take a substantial time to draw, and that ray casting is rarely supported in graphics hardware.

Spatial Enumeration [7, 2] Volume is subdivided in either a uniform or nonuniform manner. The occupancy of each object is determined for every region of space. The advantage of spatial enumeration is that volumes may be intersected in a simple manner, and the solution is view independent. The disadvantages are that large amounts of storage are required, surface information is modelled poorly, and specialised algorithms are required to render an image or interface to a standard scan-line renderer.

Scan-line Techniques [1, 10, 14] The conventional rasterisation pipeline is extended to perform clipping against the CSG tree. This is achieved either by providing a point membership classification function at low levels of the rendering pipeline, or maintaining lists of surfaces at each scanline and performing clipping over spans of pixels. The disadvantages of scan-line CSG rendering are that the solution is view dependent and that the technique is not supported in conventional graphics hardware.

Z-Buffer Algorithms [5, 6, 16, 3, 15] Surfaces are clipped in a z-buffer by a multiple pass algorithm. Arbitrary CSG trees are handled by tree normalisation and pruning techniques. [6] Methods exist to support non-convex primitives. The advantage of z-buffer algorithms is that no b-rep clipping is required, and the approach can take advantage of hardware graphics acceleration including hardware z-buffer. One disadvantage of the z-buffer approach is the dependence on z-buffer copying, which is not optimised in many hardware rendering environments.

Hybrid Algorithms [11] Clipping surfaces by using a combination of the previously mentioned techniques has the advantage that computational load is spread across CPU and graphics hardware.

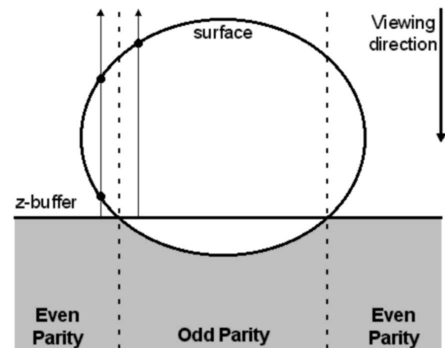


Figure 3: Z-buffer parity with respect to a surface.

The CSG rendering approach described here is a derivative z-buffer CSG rendering algorithm. In the following sections, key concepts of previous z-buffer CSG rendering techniques are presented, along with proposed extensions aimed at improving performance.

3 GOLDFEATHER CSG RENDERING

Arbitrary CSG trees can be rendered in standard z-buffered architecture using techniques described by J. Goldfeather.[6] Whilst the approach was originally developed for the Pixel-Planes graphics architecture[5, 9, 4, 8], it has recently been adapted to the OpenGL graphics environment.[16] The approach is based on the ideas of surface parity, tree normalisation, and z-buffer surface clipping. We refer to these concepts collectively as *Goldfeather CSG Rendering*. We discuss each aspect in the following subsections.

3.1 Parity

Surface parity refers to whether a surface in the z-buffer is inside or outside of a given volume. The parity of each z-buffer element can be determined by counting the number of surfaces in front of the z-buffer. This is a minor variation of the standard z-less depth testing algorithm where rather than updating the z-buffer on successful z test, a parity flag is toggled at that pixel. Regions of odd parity, where the flag is set to one, correspond to z-buffer elements volumetrically *inside* an object. Figure 3 illustrates regions of even and odd parity for a given z-buffer and surface.

The following code fragment is an OpenGL implementation, where the parity of the z-buffer is stored in the stencil buffer.

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS,0x01,0x01);
glStencilOp(GL_KEEP,GL_KEEP,GL_INVERT);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glDepthMask(GL_FALSE);
glDisable(GL_CULL_FACE);
drawSurface();
```

There are important constraints on surfaces used for parity testing. Parity logic depends on no surface being interior or exterior to the volume it represents. Also, all boundaries of the volume should be covered by surface. This means that the surface should be closed, and should not contain any holes caused by an incomplete description, such as missing polygons. Additionally, the surface must not self-intersect, forming folds or loops.

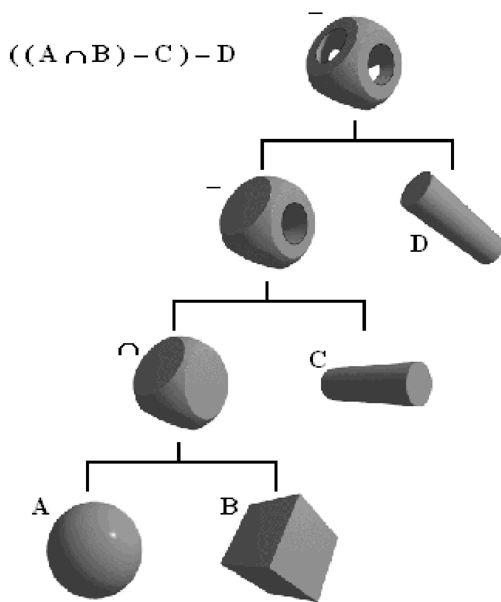
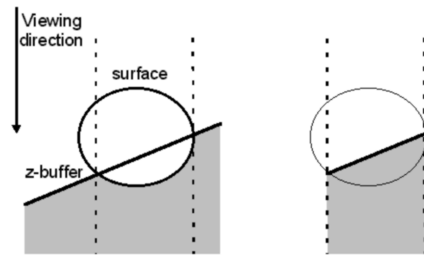
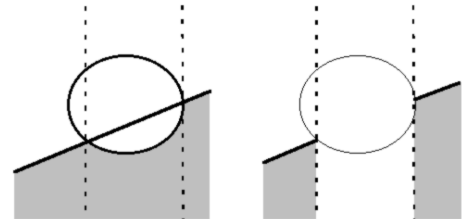


Figure 4: A Normalised CSG tree.



(a) Intersection



(b) Difference

Figure 5: Using parity to clip a z-buffer.

3.2 Tree Normalisation

Tree normalisation converts a general CSG tree to a form suitable for image space CSG rendering. A collection of primitives related by boolean union is called a sum, whilst a collection related by boolean intersection or difference is called a product. A CSG tree in sum-of-products form is said to be *normalised.* Figure 4 is a normalised CSG tree corresponding to Figure 1. Graphically, a normalised tree has three characteristics:

1. The union nodes, if any, are at the top of the tree.
2. No non-terminal node is to the right of an intersection or difference node.
3. No union node is to the left of an intersection or difference node.

The significance of tree normalisation is that it simplifies the algorithm required to render a CSG tree. Tree normalisation is the first step in a process which converts a tree into something manageable for a standard z-buffer architecture. This is because each product can be rendered by comparing primitives rather than subtrees. The union of the products is handled by sending all the products to the z-buffer with a 'z-less-than' depth test. The surfaces forming each product may be obtained in image space by using z-buffer surface clipping, discussed next.

3.3 Z-buffer Surface Clipping

Depth buffer surface clipping refers to the process of rasterising a surface into the z-buffer and detecting those pixels which satisfy the constraints of the CSG product. Since a product consists only of intersection and difference operations, each primitive may be parity tested against each other primitive in the product sequentially. The configuration of the parity test depends on the operation between the primitives. Intersection operations retain z-buffer regions of odd parity — these are z-buffer values volumetrically inside the primitive, other regions are reset to an empty state. The parity test is inverted for difference operations — regions of even parity are retained and regions of odd parity are reset. Figure 5 illustrates parity testing for intersection and difference operations.

3.4 Polygon Culling

The surface of an object can be categorised as front or back facing, in relation to the position of the viewer. In general rendering, back-facing polygons are culled since they are not visible to the user. Back-face culling depends on the surface being closed, and the polygon vertices being ordered consistently — constraints which also apply to z-buffer CSG rendering. Image space surface clipping only considers potentially visible surfaces, and these form the subset selected for z-buffer clipping. The front or back facing surface of each primitive is selected on the basis of whether the primitive is subtracted: front facing surfaces of unsubtracted primitives and the back facing surfaces of subtracted primitives.

In the following example, C and D are subtracted primitives. The CSG product is formed in the z-buffer by finding the closest clipped surface for each pixel. Figure 6 illustrates the CSG product formed by the union of clipped surfaces:

$$A \cap B - C - D$$

3.5 The Goldfeather CSG Rendering Algorithm

The Goldfeather CSG rendering algorithm is summarised below. The combined effect of tree normalisation and z-buffer surface clipping is to collapse the CSG tree into a series of clipped z-buffer surfaces which are merged by the standard 'z-less-than' depth test. One z-buffer, the *surface* z-buffer, stores the visible surface of each primitive, one at a time. This is the z-buffer where surface clipping is performed. Another z-buffer, the *output* z-buffer, is used to determine the closest clipped surface to the viewer.

```
initialise output z-buffer to zFar
for each product P do
```

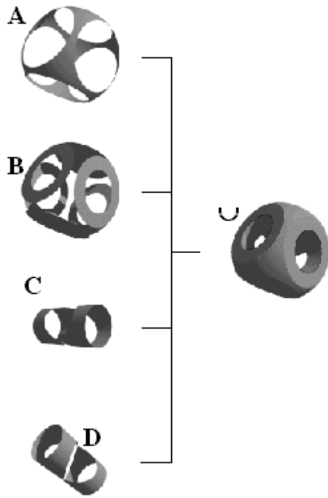


Figure 6: CSG product as union of clipped surfaces

```
for each primitive A in P do
  initialise surface z-buffer to zFar
  if A is subtracted
    draw back of A into surface z-buffer
  else
    draw front of A into surface z-buffer
  for each other primitive B in P do
      if B is subtracted
        accept pixels of even parity
      else
        accept pixel of odd parity
      apply parity test in
        surface z-buffer
  draw surface z-buffer into
      output z-buffer with z-less test
```

Goldfeather shows that the CSG rendering algorithm can be extended to non-convex primitives.[6] Optimisation of the normalisation algorithm by means of tree pruning is also discussed. The requirements of implementing this approach in OpenGL are a single colour buffer, a single z-buffer, a stencil buffer and the ability to save and restore the contents of the z-buffer[16].

4 TAKING ADVANTAGE OF DEPTH COMPLEXITY

In analysing performance issues it is useful to talk about a single product of length n. Figure 7 illustrates the steps required to subtract four spheres from a rectangular block with the Goldfeather approach. Each row corresponds to a primitive clipping operation, which are essentially independent operations. In the first column, the primitive surface is drawn into the z-buffer. In the second column, the z-buffer surface is clipped against other primitives in the product. In the third column, the clipped surface is merged into the final result. Each row requires n primitive rasterisations (where n is the number of primitives) and one z-buffer copy operation.

Our proposed improvement to the algorithm is to reduce the number of passes by clipping more than one primitive at a time. In Figure 8 the spheres are clipped together, rather than in seperate passes, taking advantage of the fact that the spheres no do not overlap in image space. The term *depth complexity* refers to the number of primitives which cover each pixel. We denote k as being the maximum depth complexity of a pixel given a specific viewing direction. We propose clipping on the basis of *layers* rather than primitives — the number of required layers being k.

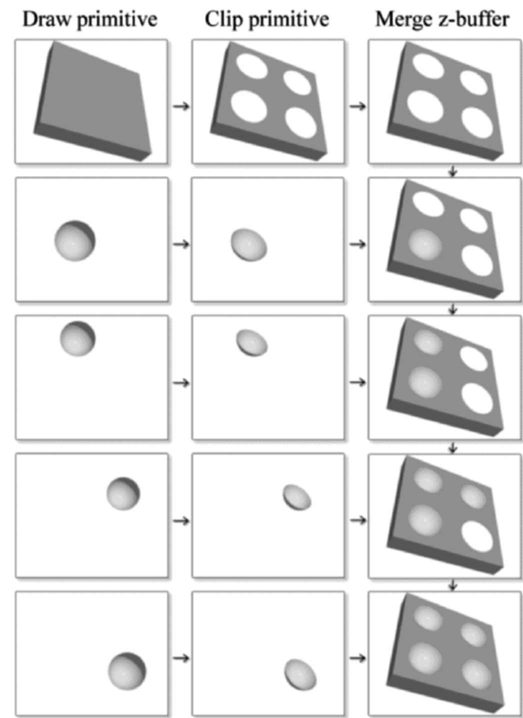


Figure 7: Clipping on per-primitive basis.

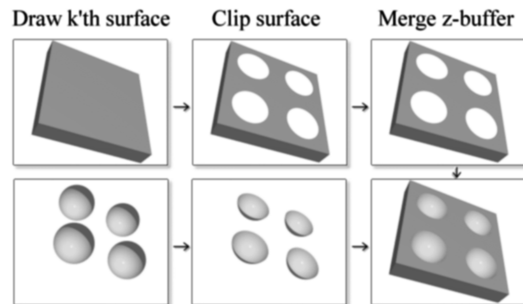


Figure 8: Clipping on per-layer basis.

This modification to the algorithm introduces three new subproblems. One is to determine k, the depth complexity of the product. Another is to extract each layer from a given set of primitives. The final problem is to ensure reliable clipping of surfaces against their corresponding primitives. We will discuss each of these issues seperately.

4.1 Determining k

A simple stencil test can be used to determine k. The following OpenGL code fragment finds the number of primitives that cover each pixel, and stores this count in the stencil buffer. The number of clipping passes required by the algorithm is then determined by

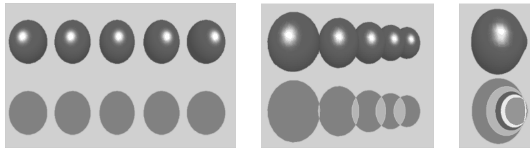


Figure 9: Image space depth complexity

finding the maximum stencil buffer value.

```
glClear(GL_STENCIL_BUFFER_BIT);
glDepthMask(GL_FALSE);
glDepthFunc(GL_ALWAYS);
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS,0xff,0xff);
glStencilOp(GL_INCR,GL_INCR,GL_INCR);
drawPrimitiveSurfaces();
```

The result of this operation depends on the viewing direction, as illustrated in Figure 9. Five spheres are viewed from different directions and the counts in the stencil buffer visualised via different shades. The first scenario where no primitives overlap is ideal for clipping, since all five spheres may be clipped at once. The third scenario is the worst possible case, since all five spheres overlap at least one pixel. For this viewing direction, clipping primitives as a sequence of five layers, rather than five primitives offers no speedup over the standard algorithm.

4.2 Layer Extraction

A stencil test may be employed to draw only the n'th layer of a set of primitives. The following OpenGL code fragment increments a count in the stencil buffer for each pixel covered by each primitve. The primitive is only drawn to the z-buffer if the stencil count equals the number of the desired layer.

```
glClear(GL_STENCIL_BUFFER_BIT);
glDepthMask(GL_TRUE);
glDepthFunc(GL_ALWAYS);
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_EQUAL,n,0xff);
glStencilOp(GL_INCR,GL_INCR,GL_INCR);
drawPrimitiveSurfaces();
```

It should be noted that the particular surfaces in each layer are determined by the order in which primitives are presented to OpenGL. The first layer will consist of surface from the first primitive, and portions of the second primitive not overlapping the first, and so on.

4.3 Layer Clipping

The Goldfeather CSG rendering algorithm implicitly avoids clipping z-buffer surfaces against the corresponding primitive. Clipping on a per-layer basis however, requires a clipping algorithm which will behave in this situation sensibly. Specifically, the front facing surface of an intersecting primitive should be classified as 'inside' the primitive. Similarly, the back facing surface of a subtracted primitive should be classified as 'outside'. Configuring the parity test to utilise the 'z-less-or-equal' z-test satisfies these constraints, although they do depend on triangles being rasterised to precisely the same z-buffer values on different passes.

5 THE NEW ALGORITHM

The new algorithm is summarised below. Two z-buffers are utilised in a similar manner to the Goldfeather algorithm. The *surface* z-buffer is used to extract and clip each layer of surfaces in the product. The *output* z-buffer accumulates the final result, taking the closest pixel from each instance of the surface z-buffer. Like the Goldfeather algorithm, a final pass over all primitives draws the correct colour into the frame buffer, using a 'z-equal-test' on the final z-buffer.

```
initialise output z-buffer to zFar
for each product P do
  for each layer k in P do
    initialise surface z-buffer to zFar
    draw k'th surface into surface z-buffer
    for each primitive A in P do
        if A is subtracted
          accept pixels of even parity
        else
          accept pixel of odd parity
        apply parity test in
          surface z-buffer
    draw surface z-buffer into
        output z-buffer with z-less test
```

5.1 Comparative Analysis

The Goldfeather algorithm performs one clipping operation per primitive. The time required for each clipping operation is $an + b$, where a is related to polygon rasterisation speed, and b is related to z-buffer copy speed. The time required to draw the product is: $an^2 + bn$.

The modified CSG rendering algorithm requires rendering n primitives to extract a layer, rendering n primitives to clip the layer, and one z-buffer copy operation per layer. Multiplying this cost by the number of layers k, the time required to draw a product is: $2akn + bk$.

The important difference between this and the standard algorithm is the behavior of k. The depth complexity of a scene is determined by the viewing direction and the particular configuration of primitives in the product. The usefulness of this approach is therefore limitied to those applications where $k < n$. For situations where $k \approx n$, the performance is expected to be similar to that of the standard algorithm — $O(n^2)$. It should be noted that since layer extraction doubles the polygon rasterisation load, the standard algorithm should be faster when $k = n$.

5.2 Further Work

It is expected that this improvement to the Goldfeather CSG rendering algorithm will be of particular use in the simulation of manufacturing processes. Our observation is that this application domain has the characteristic that k tends to be significantly less than n. A systematic investigation would be required to quantify this relationship.

The constants referred to as a and b could be also be quantified. These correspond to rasterisation and z-buffer copying performance respectively. Our observation is that z-buffer copy rate is the bottleneck on platforms we have experimented with. An investigation to quantify these characteristics would be useful in predicting the usefulness of z-buffer CSG algorithms for different problem domains. This would also aid the identification of suitable platforms for hardware implementation of the algorithm.

Implementation of the algorithm could include a decision making policy to choose the best clipping strategy, given n and k.

In certain circumstances, clipping on a primitive basis is optimal, while in other circumstances, layer clipping would be more efficient.

6 CONCLUSION

We have described an image-space CSG subtraction algorithm which performs better in many useful cases. Modifying the Goldfeather CSG rendering algorithm to exploit depth complexity offers O(kn) performance rather than O(n^2). The algorithmic implication of this modification is to increase the cost of the inner loop, while potentially reducing the number of iterations of the outer loop.

A significant implication of this result, given current graphics hardware architectures, is the reduced z-buffer copy bandwidth and z-comparison requirement. This is reduced from n to k.

7 ACKNOWLEDGEMENTS

The following people are acknowledged for their suggestions, guidance and contributions to this research: Dr. Mike Simakov and the software development team at ANCA Pty. Ltd. This research is supported by the Co-operative Research Center for Intelligent Manufacturing Systems & Technologies, their industry partner ANCA Pty. Ltd., and their academic partner RMIT University.

References

[1] P. Atherton, "A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry", *Computer Graphics (Proc Siggraph)*, Vol. 17, No. 3, July 1983, pp. 73-82

[2] J. Ayala, P. Brunet, R. Juan, I. Navazo "Object Representation by Means of Nonminimal Division Quadtrees and Octrees", *ACM Trans. on Graphics*, Vol. 4, No. 1, Jan. 1985, pp. 41-59

[3] D. Epstein, F. Jansen, J. Rossignac, "Z-Buffer Rendering from CSG: The Trickle Algorithm", *IBM Research Report RC 15182*, Nov. 1989

[4] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, L. Westover "PixelFlow: The Realization" *Proc. 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, August 1997, pp. 3-13

[5] J. Goldfeather, J. Hultquist, H. Fuchs, "Fast Constructive Solid Geometry in the Pixel-Powers Graphics System", *Computer Graphics (Proc. Siggraph)*, Vol. 20, No. 4, Aug. 1986, pp. 107-116

[6] J. Goldfeather, S. Molnar, G. Turk, H. Fuchs, "Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning", *IEEE CG&A*, Vol. 9, No. 3, May 1989, pp. 20-28.

[7] G. Hunter, K. Steiglitz "Operations on Images Using Quad Trees", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-1, No. 2, Apr. 1979, pp. 145-153

[8] S. Molnar "Combining Z-buffer Engines for Higher-Speed Rendering" *Proc. Eurographics '88, Third Workshop on Graphics Hardware*, Sep. 1998, pp. 171-182

[9] S. Molnar, J. Eyles, J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition" *Siggraph 92*, pp. 231-240

[10] N. Okino, Y. Kakazu, M. Morimoto, "Extended Depth-Buffer Algorithms for Hidden-Surface Visualization", *IEEE CG&A*, Vol. 4, No. 5, May 1984, pp. 79-88

[11] A. Rappoport, S. Spitz, "Interactive Boolean Operations for Conceptual Design of 3-D Solids", *Siggraph 97*, pp. 269-278

[12] A. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems", *Computing Surveys*, Vol. 12, No. 4, Dec. 1980, pp. 437-464

[13] A. Requicha, H. Voelcker, "Boolean Operations in Solid Modelling: Boundary Evaluation and Merging Algorithms", *Proc. of the IEEE*, Vol. 73, No. 1, Jan. 1985, pp. 30-44

[14] J. Rossignac, A. Requicha, "Depth-Buffering Display Techniques for Constructive Solid Geometry", *IEEE CG&A*, Vol. 6, No. 9, Sept. 1986, pp. 29-39

[15] J. Rossignac, J. Wu "Correct Shading of Regularized CSG solids using a Depth-Interval Buffer", *Proc. Fifth Eurographics Workshop on Graphics Hardware*, 1990.
Also in: Grimsdale, R.L., Kaufman A., (eds), *Advances in Computer Graphics Hardware V*, Sprinter-Verlag, Berlin, 1990, pp. 117-138

[16] T. F. Wiegand "Interactive Rendering of CSG Models" *Computer Graphics Forum*, Vol. 15, No. 4, Oct. 1996, pp. 249-261