Volume 15, (1996) number 4 pp. 249–261

Interactive Rendering of CSG Models

T. F. Wiegand†

The Martin Centre for Architectural and Urban Studies
The University of Cambridge, Cambridge, UK

Abstract

We describe a CSG rendering algorithm that requires no evaluation of the CSG tree beyond normalization and pruning. It renders directly from the normalized CSG tree and primitives described (to the graphics system) by their facetted boundaries. It behaves correctly in the presence of user defined, "near" and "far" clipping planes. It has been implemented on standard graphics workstations using Iris GL [1] and OpenGL [2] graphics libraries. Modestly sized models can be evaluated and rendered at interactive (less than a second per frame) speeds. We have combined the algorithm with an existing B-rep based modeller to provide interactive rendering of incremental updates to large models.

1. Introduction

Constructive Solid Geometry (CSG) within an interactive modelling environment provides a simple and intuitive approach to solid modelling. In conventional modelling systems primitives are first positioned, a boolean operation is performed and the results then rendered. Often the correct position cannot be gauged easily from display of the primitives alone. A sequence of trial and error may be initiated or perhaps a break from the normal modelling process to calculate the correct position numerically. Conceptual modelling is inhibited — usually a design is fully fledged before modelling commences. Interactive rendering offers the promise of a modelling system where designers can easily explore possibilities within the CSG paradigm. For instance, a designer could drag a hole defined by a complex solid through a workpiece, observing the new forms that emerge.

Interactive rendering of CSG models has previously been implemented with special purpose hardware [3, 4, 5]. We believe that such systems should be based on an existing, commonly available graphics library. Use of an existing graphics library simplifies development, protects investment in proprietary graphics hardware, and leverages off future improvements in the hardware supported by the library. Conversion of the CSG tree for a model into a boundary representation (B-rep) meets this goal but is typically too slow for interactive modification.

The surfaces in the B-rep of a model are a subset of the surfaces of the primitives in the CSG tree for the model. Conversion to a B-rep is then the classification of the surfaces of each primitive into portions that are "inside", "outside", or "on" the surface of the fully evaluated model. Display of the model only requires classification of the points on the surfaces which project to each pixel. Point classification is much simpler than surface classification. Geometrically, point classification requires intersection of the primitives with rays through each pixel, while surface classification requires intersection of the primitive surfaces with each other.

Thibault and Naylor [6] describe a surface classification based approach. They build BSP trees for each primitive and perform the classification by merging the trees together. The resulting tree is equivalent to a BSP tree built from the B-rep of the model. The complete evaluation process is too slow for interactive rendering. They describe an incremental version of their algorithm which provides interactive rendering speeds within a modelling environment.

There are variations of most rendering algorithms which use point classification. These include ray trac-

† Supported by Informatix, Inc. Tokyo.

ing [7], scan line methods [8], and depth-buffer methods [9, 10, 11]. Much attention has been focused on optimising point classification for this purpose [12]. These algorithms all add point classification within the lowest levels of the standard algorithms. We require an algorithm which can be implemented using an existing graphics library.

Goldfeather, Molnar, Turk and Fuchs [13] describe an algorithm that first normalizes a CSG tree before rendering the normalized form. It operates in a SIMD pixel parallel way on an augmented frame buffer (Pixel-planes 4) which has two depth (Z) buffers, two color buffers and flag bits per pixel. We have developed a new version of this algorithm capable of being implemented using an existing graphics library on a conventional graphics workstation. Our algorithm requires a single depth buffer, single color buffer, stencil (flag bits) buffer and the ability to save and restore the contents of the depth buffer.

In section 2 we review the algorithm described by Goldfeather et. al. [13]. We have restructured the presentation of the ideas to make them more amenable to implementation on a conventional graphics workstation. Our implementation is described in section 3. In section 4 we describe the integration of user defined, "near" and "far" clipping planes into the algorithm. In section 5 we describe use of the algorithm within an interactive modelling system. The system maintains fully evaluated B-rep versions of models and uses the rendering algorithm for interactive changes to the models. Section 6 presents performance statistics for our current implementation using the Silicon Graphics GL library [1].

2. Rendering a CSG tree using pixel parallel operations

We would advise interested readers to refer to Goldfeather et. al. [13] for a fuller description of the algorithm which we summarize in this section.

A CSG tree is either a primitive or a boolean combination of sub-trees with intersection($\cap$), subtraction($-$) or union($\cup$) operators. A CSG tree is in normal (sum of products) form when all intersection or subtraction operators have a left subtree which contains no union operators and a right subtree that is simply a primitive. For example $(((A \cap B)-C) \cup (D \cap (E-(F \cap G)))) \cup H$, where A–H represent primitives, is in normal form. We shall assume left association of operators so the previous expression can be written as $(A \cap B-C) \cup (D \cap E-F \cap G) \cup H$. This expression has three products. The primitives A, B, D, E, G, H are *uncomplemented*, C and F are *complemented*.

The normalization process recursively applies a set of production rules to a CSG tree which use the associative and distributive properties of boolean operations. Determining an appropriate rule and applying it uses only local information (type of current node and child node types). The production rules and algorithm used are :

1. $X-(Y \cup Z) \rightarrow (X-Y)-Z$
2. $X \cap (Y \cup Z) \rightarrow (X \cap Y) \cup (X \cap Z)$
3. $X-(Y \cap Z) \rightarrow (X-Y) \cup (X-Z)$
4. $X \cap (Y \cap Z) \rightarrow (X \cap Y) \cap Z$
5. $X-(Y-Z) \rightarrow (X-Y) \cup (X \cap Z)$
6. $X \cap (Y-Z) \rightarrow (X \cap Y)-Z$
7. $(X-Y) \cap Z \rightarrow (X \cap Z)-Y$
8. $(X \cup Y)-Z \rightarrow (X-Z) \cup (Y-Z)$
9. $(X \cup Y) \cap Z \rightarrow (X \cap Z) \cup (Y \cap Z)$

```
proc normalize(T : tree)
{
    if T is a primitive {
      return
    }
    repeat {
        while T matches a rule from 1–9 {
            apply first matching rule
        }
        normalize(T.left)
    } until (T.op is a union) or
        ((T.right is a primitive) and
         (T.left is not a union))
    normalize(T.right)
}
```

Goldfeather et. al. [13] show that the algorithm terminates, generates a tree in normal form and does not add redundant product terms or repeat primitives within a product.

Normalization can add many primitive leaf nodes to a tree with a possibly exponential increase in tree size. In most cases, a large number of the products generated by normalization play no part in the final image, because their primitives do not intersect. A limited amount of geometric information (bounding boxes of primitives) is used to prune CSG trees as they are normalized. Bounding boxes are computed for each operator node using the rules :

1. $\text{Bound}(A \cup B) = \text{Bound}(\text{Bound}(A) \cup \text{Bound}(B))$
2. $\text{Bound}(A \cap B) = \text{Bound}(\text{Bound}(A) \cap \text{Bound}(B))$
3. $\text{Bound}(A-B) = \text{Bound}(A)$

Here A and B are arbitrary child nodes. After each step of the normalization algorithm the tree is pruned by applying the following rules to the current node :

1. $A \cap B \rightarrow \emptyset$, if Bound($A$) does not intersect Bound(B).
2. $A - B \rightarrow A$, if Bound(A) does not intersect Bound(B).

Normalization of the tree allows simplification of the rendering problem. The union of two or more solids can be rendered using the standard depth (Z) buffer hidden surface removal algorithm used by most graphics workstations. The rendering algorithm needs only to render the correct depth and color for each product in the normalized CSG tree and then allow the depth buffer to combine the results for each product.

Each product can be rendered by rendering each visible surface of a primitive and trimming (intersecting or subtracting) the surface with the remaining primitives in the product. The visible surfaces are the front facing surfaces of uncomplemented primitives and the back facing surfaces of complemented primitives. This observation allows a further rewriting of the CSG tree where each product is split into a sum of *partial products*. A convex primitive has one pair of front and back surfaces per pixel. A non-convex primitive may have any number of pairs of front and back surfaces per pixel. A k-convex primitive is defined as one that has at most k pairs of front and back surfaces per pixel from any view point. We shall use the notation A_k to represent a k-convex primitive and A_{fn} to represent the nth front surface (numbered 0 to $k-1$) of primitive A_k and A_{bn} to represent the nth back surface of A_k. In the common case of convex primitives, we shall drop the numerical subscripts. Thus, $A-B$ expands to $(A_f - B) \cup (B_b \cap A)$ in sum of partial products form; while $A_2 - B$ expands to $(A_{f0} - B) \cup (A_{f1} - B) \cup (B_b \cap A_2)$. We call the primitive whose surface is being rendered the *target* primitive of the partial product. The remaining primitives are called *trimming* primitives.

The sum of partial products form again simplifies the rendering problem. It is now reduced to correctly rendering partial products before combining the results with the depth buffer. Additional *difference* pruning may also be carried out when products have been expanded to partial products :

3 $A_b \cap B \rightarrow \emptyset$, if Bound($A$) does not intersect Bound(B).

A partial product is rendered by first rendering the target surface of the partial product. Each pixel in the surface is then classified in parallel against each of the trimming primitives. To be part of the partial product surface, each pixel must be *in* with respect to any uncomplemented primitives and *out* with respect to any complemented ones. Those pixels which do not meet these criteria are trimmed away (colour set to background, depth set to initial value).

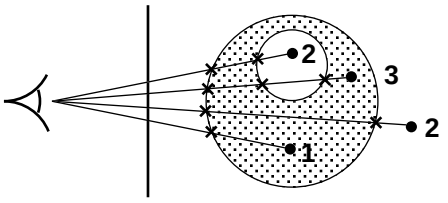

Figure 1: *Classifying per pixel depth values against a primitive*

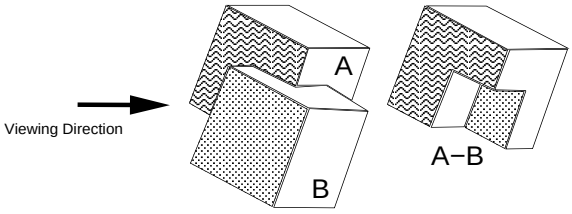

Figure 2: *A simple CSG expression*

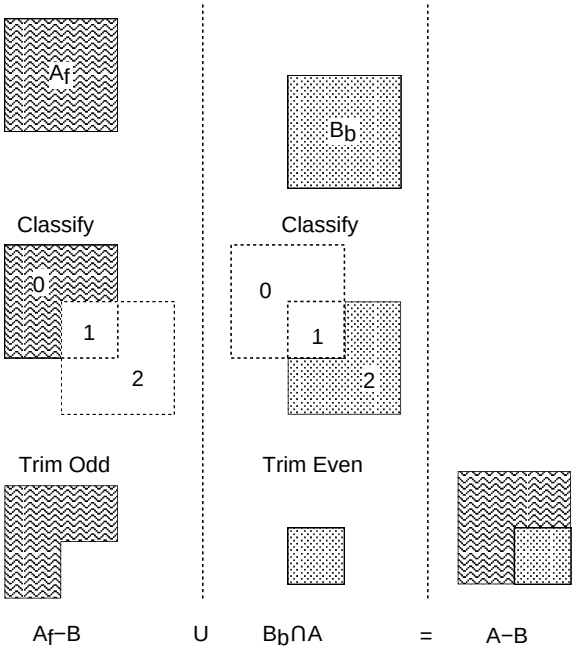

Figure 3: *Rendering figure 2 as two partial products*

Primitives must be formed from closed (possibly nested) facetted shells. Pixels can then be classified against a trimming primitive by counting the number of times a primitive fragment is closer during scan conversion of the primitive's faces. If the result is odd the pixel is *in* with respect to the primitive (figure 1). Pixels can be classified in parallel by using a 1 bit flag per pixel whose value is toggled whenever scan conversion of a trimming primitive fragment is closer than the pixel's depth value.

Figure 3 illustrates the process for $A - B$ looking along the view direction shown in figure 2. First, A_f is rendered, classified against B and trimmed ($A_f - B$). Then B_b is rendered, classified against A and trimmed ($B_b \cap A$). Finally, the two renders are composited together.

Rendering the appropriate surface of a convex primitive is simple as there is only one pair of front and back surfaces per pixel. Most graphics libraries support front and back face culling modes. To render all possible surfaces of an arbitrary k-convex primitive separately requires a $\log_2 k$ bit count per pixel. To render the jth front (or back) facing surface of a primitive, the front (or back) facing surfaces are rendered incrementing the count for each pixel and only enabling writes to the colour and depth buffers for which the count is equal to j.

3. Implementation on a conventional graphics workstation

The algorithm described in section 2 maps naturally onto a hardware architecture which can support two depth buffers, two colour buffers and a stencil buffer. One pair of depth and colour buffers, together with the stencil buffer, are used to render each partial product. The results are then composited into the other pair of buffers. Unfortunately, conventional graphics workstation hardware typically supports only one depth buffer. One approach is to use the hardware provided depth, colour and stencil buffers to render partial products; retrieving the results from the hardware and compositing in local workstation memory. The final result can then be returned direct to the frame buffer. This approach does not make the best use of the workstation hardware. Modern hardware tends to be highly pipelined. Interrupting the pipeline to retrieve results for each partial product will have a considerable performance penalty. In addition, the hardware is typically optimized for flow of data from local memory, through the pipeline and into the frame buffer. Data paths from the frame buffer back to local memory are likely to be slow, especially given the volume of data to be retrieved compared to the compact instructions given to the hardware to draw the primitives. Finally, the compositing operation in local memory will receive no help from the hardware.

Our approach attempts to extract the maximum benefit from any graphics hardware by minimizing the traffic between local memory and the hardware and by making sure that the hardware can be used for all rendering and compositing operations. The idea is to divide the rendering process into two phases — classification and final rendering. Before rendering begins the current depth buffer contents are saved into local memory. We then classify each partial product surface in turn. An extra stencil buffer bit (*accumulator*) per surface stores the results of the classification. During this process updates to the colour buffer are disabled. Once classification is complete, we restore the depth buffer to the saved state and enable updates to the colour buffer. Finally, each partial product surface is rendered again using the stored classification results as a mask (or stencil) to control update of the frame buffer. At the same time the depth buffer acts to composite the pixels which pass the stencil test with those already rendered.

The number of surfaces for which we can perform classification is limited by the depth of the stencil buffer. If the capacity of the stencil buffer is exceeded the surfaces must be processed in multiple passes with the depth buffer saved and restored during each pass. We can reduce the amount of data that needs to be copied by only saving the parts of the depth buffer that will be modified by classification during each pass. The first pass of each frame does not need to save the depth buffer at all as the values are known to be those produced by the initial clear. Instead of restoring, the depth buffer is cleared again. Thus, for simple models rendered at the start of a frame, no depth buffer save and restore is needed at all.

A surface may appear in more than one partial product in the normalized CSG tree. We exploit this by using the same accumulator bit for all partial products with the same surface. Classification results for each partial product are ORed with the current contents of the accumulator.

The stencil bits are partitioned into count bits (S_{count}), a parity bit (S_p) and an accumulator bit (S_a) per surface. $\log_2 k$ count bits are required where k is the maximum convexity of any primitive with a surface being classified in the current pass. The count and parity bits are used independently and may be overlapped. Table 1 shows the number of stencil buffer bits required to classify and render a single surface for primitives of varying convexity. The algorithm requires an absolute minimum of 2 bits for 1-convex and 2-convex primitives, classifying and rendering a single surface in a pass. In practice nearly all primitives used

Convexity	1	2	3–4	5–8	9–16	17-32	33-64	65–128
S_p	1	1	1	1	1	1	1	1
S_{count}	0	1	2	3	4	5	6	7
S_p and S_{count}	1	1	2	3	4	5	6	7
With 1 accumulator (S_0)	2	2	3	4	5	6	7	8
With 3 accumulators ($S_{0..2}$)	4	4	5	6	7	8	9	10
With 7 accumulators ($S_{0..6}$)	8	8	9	10	11	12	13	14

Table 1: *Stencil buffer usage with primitive convexity*

in pure CSG trees are 1-convex. With 8 stencil bits the algorithm can render from 7 1-convex primitives, to 1 surface of a 128-convex primitive, in a single pass.

Partial products are gathered into groups such that all the partial products in a group can be classified and rendered in one pass. The capacity of a group is defined as the number of different target surfaces that partial products in the group may contain. Capacity is dependent on the stencil buffer depth and the greatest convexity of any of the target primitives in the group (table 2). Groups are formed by adding partial products in ascending order of target primitive convexity. Once one partial product with a particular target surface is added, all others with the same target surface can be added without using any extra capacity. Adding a partial product with a higher convexity than any already in the group will reduce the group capacity. If there is insufficient capacity to add the minimum convexity remaining partial product, a new group must be started.

Each group is processed in a separate pass in which all target surface primitives are classified and then rendered. Frame buffer wide operations are limited to areas defined by the projection of the bounding box of the current group or partial product. We present pseudo-code for the complete rendering process below. The procedures "glPrim(prim, tests, buffers, ops, pops)" and "glSet(value, tests, buffer, ops, pops)" should be provided by the graphics library. The first renders (scan converts) a primitive where "tests" are the tests performed at each pixel to determine if it can be updated, "buffers" specifies the set of buffers enabled for writing if the "tests" pass (where C is colour, Z is depth and S is stencil), "ops" are operations performed on the stencil bits at each pixel in the primitive, and "pops" are operations to be performed on the stencil bits at each pixel only if "tests" pass. The second procedure is similar but attempts to globally set values for all pixels. Iris GL [1] and OpenGL [2] are two graphics libraries which provide equivalents to the glPrim and glSet procedures described here. We use the symbol Z_P to denote the depth value at a pixel due to the scan conversion of a primitive, P. Hence, "$Z_P < Z$" is the familiar Z buffer hidden surface removal test. We use Z_f to represent the furthest possible depth value.

```
glSet(0, ALWAYS, S, ∅, ∅)
glSet("far", ALWAYS, Z, ∅, ∅)
for first group G {
    classify(G)
    glSet(Z_f, ALWAYS, Z, ∅, ∅)
    renderGroup(G)
} for each subsequent group G {
      save depth buffer
      glSet(Z_f, ALWAYS, Z, ∅, ∅)
      classify(G)
      restore depth buffer
      renderGroup(G)
}

proc classify(G : group)
{
      a = 0
      for each target surface B in G {
          for each partial product R {
              renderSurface(B)
              for each trimming primitive P in R {
                  trim(P)
              }
              glSet(1, S_a = 0 & Z ≠ Z_f, S_a, ∅, ∅)
              glSet(Z_f, ALWAYS, Z, ∅, ∅)
          }
          a = a + 1
      }
}

proc renderGroup(G : group)
{
      a = 0
      for each target primitive P in G {
          glPrim(P, S_a = 1 & Z_P < Z, C & Z, ∅, ∅)
          glSet(0, ALWAYS, S_a, ∅, ∅)
          a = a + 1
      }
}
```

Capacity		Maximum Target Primitive Convexity							
		1	2	3–4	5–8	9–16	17-32	33-64	65–128
Stencil Buffer Depth	2	1	1	-	-	-	-	-	-
	3	2	2	1	-	-	-	-	-
	4	3	3	2	1	-	-	-	-
	5	4	4	3	2	1	-	-	-
	6	5	5	4	3	2	1	-	-
	7	6	6	5	4	3	2	1	-
	8	7	7	6	5	4	3	2	1

Table 2: *Group Capacity*

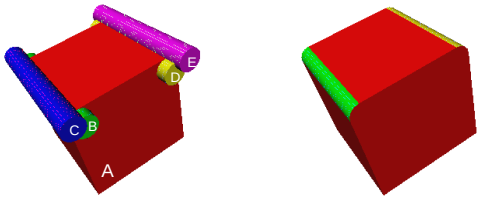

Figure 4: *(a) Primitives, (b) Rendering* $(A \cap B \cup A - C) \cap (A \cap D \cup A - E)$

```
proc renderSurface(B : surface)
{
      P = target primitive containing B
      n = surface number of B
      k = convexity of P
      if P is uncomplemented {
        enable back face culling
      } else {
        enable front face culling
      }
      if k = 1 {
        glPrim(P, ALWAYS, Z, ∅, ∅)
      } else  {
        glPrim(P, S_count = n, Z, inc S_count, ∅)
        glSet(0, ALWAYS, S_count, ∅, ∅)
      }
}

proc trim(P : primitive)
{
      glPrim(P, Z_P < Z, ∅, ∅, toggle S_p)
      if P is uncomplemented {
        glSet(Z_f, S_p = 0 , Z, ∅, ∅)
      } else {
        glSet(Z_f, S_p = 1 , Z, ∅, ∅)
      }
      glSet(0, ALWAYS, S_p, ∅, ∅)
}
```

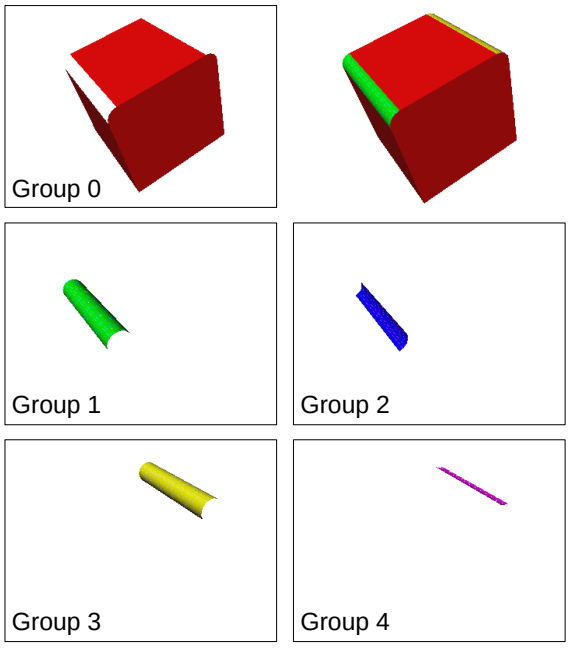

Figure 5: *Rendering each product group separately*

Figure 4 shows five primitives and a rendered CSG tree of the primitives. The expression $((A \cap B) \cup (A - C)) \cap ((A \cap D) \cup (A - E))$ normalizes to $(A \cap B \cap D) \cup (A \cap D - C) \cup (A \cap B - E) \cup (A - C - E)$. Expanding to partial products and grouping gives :

0: $(A_f \cap B \cap D) \cup (A_f \cap D - C) \cup (A_f \cap B - E) \cup (A_f - C - E)$
1: $(B_f \cap A \cap D) \cup (B_f \cap A - E)$
2: $(C_b \cap A \cap D) \cup (C_b \cap A - E)$
3: $(D_f \cap A \cap B) \cup (D_f \cap A - C)$
4: $(E_b \cap A \cap B) \cup (E_b \cap A - C)$

Figure 5 shows the result of rendering each product group separately. Product groups 2 and 4 are not

visible in the combined image as they are behind the surfaces from groups 1 and 3.

4. Clipping planes and half spaces

Interactive inspection of solid models is aided by means of clipping planes which can help reveal internal structure. After a clipping plane has been defined and activated all subsequently rendered geometry is clipped against the plane and the parts on the *out* side discarded. The rendering of solids as closed shells means that clipping will erroneously reveal the interior of a shell when a portion of the shell is clipped away. Rossignac, Megahed and Schneider [14] describe a stencil buffer based technique for "capping" shells where they intersect a clipping plane. Their algorithm will also highlight interferences (intersections) between solids on the clipping plane.

Clipping a solid and then capping is equivalent to intersection with a half space. We can trivially render an intersection between a solid S and a halfspace H by constructing a convex polygonal primitive P where one face lies on the plane defining H and has edges which do not intersect the bounding box of S. The other faces of P should not intersect S at all. Rendering $S \cap P$ is equivalent to rendering the solid defined by $S \cap H$.

Rossignac, Megahed and Schneider's [14] capping algorithm can be easily integrated with our algorithm to make use of auxiliary clipping planes in rendering CSG trees involving halfspaces. As a halfspace is infinite we assume that it will always be intersected with a finite primitive in any CSG expression. Note that $S - H$ is equivalent to $S \cap \overline{H}$ where $\overline{H}$ is simply H with the normal of the halfspace defining plane reversed.

A halfspace acts as a trimming primitive by activating a clipping plane for the halfspace during the rendering of the target primitive. The stencil buffer is unused. The set of halfspaces in a product can be considered as a 1-convex target primitive. Its surface can be rendered by rendering the defining plane (or rather a sufficiently large polygon lying on the plane) of each halfspace while clipping planes are active for each of the other halfspaces. Each clipping plane is deactivated while it is being rendered to prevent it from clipping itself.

```
proc render(H : halfspace set)
{
    for each defining plane P of H {
        Activate clipping plane defined by P
    }
    for each front facing defining plane P of H {
        Deactivate clipping plane defined by P
        renderPlane(P)
        Activate clipping plane defined by P
    }
    for each defining plane P of H {
        Deactivate clipping plane defined by P
    }
}
```

This approach has three advantages over rendering halfspaces as normal primitives. Firstly, the halfspace set only has to be rendered as a target primitive, all trimming by halfspaces uses the clipping planes. Secondly, each target primitive is clipped, reducing the amount of data written to the frame buffer at the cost of the extra geometry processing required by clipping. Thirdly, a solid/halfspace intersection can be correctly rendered using the algorithm for 1-convex solids ($k = 1$), independent of actual primitive convexity.

Rendering a k-convex target primitive using the algorithm for 1-convex solids results in the nearest surface being drawn (with depth buffering active). The nearest surface (*after* clipping) of a concave primitive will be visible in the intersection with a half space. Rendering an arbitrary CSG tree using the 1-convex algorithm will render the result of evaluating the CSG description on the "nearest spans" (nearest front to nearest back facing surface for each pixel) of each primitive. For interactive use the nearest spans are often all we are interested in. If not, then clipping planes may be used to delimit regions of interest within which the nearest spans will be correctly rendered. Thus, a lower cost, reduced quality mode of rendering is also available.

In addition to user defined clipping planes, all geometry is usually clipped to "near" and "far" planes. These planes are perpendicular to the viewing direction. All geometry must be further from the eye position than the near plane and nearer than the far plane. The near and far planes also define the mapping of distances from the eye point to values stored in the depth buffer. Points on the near plane map to the minimum depth buffer value and points on the far plane map to the maximum depth buffer value. The algorithm described in section 3 will fail if any primitive is clipped by either the near or far clipping plane.

In practice the far clipping plane can always be safely positioned beyond the primitives. The near plane is more troublesome. Firstly, it cannot be positioned behind the eye point. Secondly, the resolution of the depth buffer is critically dependent on the position of the near clip plane. It should be positioned as far from the eye point as possible. Consider rendering $A - B$ and positioning the eye in the hole in A formed by subtracting B. Near plane clipping is unavoidable. We can extend our algorithm to cap trimming prim-

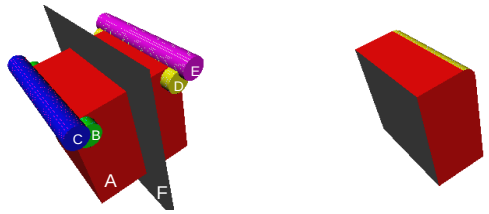

Figure 6: *(a) Primitives, (b) Rendering* $(A \cap B \cup A - C) \cap (A \cap D \cup A - E) \cap F$

itives if they will be subject to near plane clipping. Clipping of target primitives is not a problem unless the eye point is positioned inside the evaluated CSG model.

The trimming primitive is rendered twice while toggling S_p; firstly, with the depth buffer test disabled; secondly, with the depth buffer test enabled. The first render sets the parity bit where capping is required. The second completes the classification as above.

proc trim(P : primitive)
{
glPrim(P, ALWAYS, $\emptyset$, $\emptyset$, toggle S_p)
glPrim(P, $Z_P < Z$, $\emptyset$, $\emptyset$, toggle S_p)
if P is uncomplemented {
glSet(Z_f, $S_p = 0$, Z, $\emptyset$, $\emptyset$)
} else {
glSet(Z_f, $S_p = 1$, Z, $\emptyset$, $\emptyset$)
}
glSet(0, ALWAYS, S_p, $\emptyset$, $\emptyset$)
}

Figure 6 shows our earlier example intersected with a single clipping plane / half space. The normalized CSG description is $(A \cap B \cap D \cap F) \cup (A \cap D \cap F - C) \cup (A \cap B \cap F - E) \cup (A \cap F - C - E)$.

The normalization and pruning algorithm described in section 2 needs to be extended to cope with half-space primitives. The extensions required are in the form of additional rules for bounding box generation, normalization and pruning (H is a halfspace) :

Bounding Box Generation

4. Bound$(A \cap H)$ = Bound(A)

Normalization

0. $X - H \rightarrow X \cap \overline{H}$

Pruning

4. $A \cap H \rightarrow \emptyset$, if Bound$(A)$ is outside H.
5. $A \cap H \rightarrow A$, if Bound(A) is inside H.
6. $A \cap H - B \rightarrow A \cap H$, if Bound$(B)$ is outside H.
7. $A_b \cap H \rightarrow A_b$, if Bound$(A)$ is inside H.
8. $H_f - A \rightarrow H_f$, if Bound(A) does not intersect H.

Our earlier example (figure 6) contains many pruning possibilities. The normalized CSG tree is $(A \cap B \cap D \cap F) \cup (A \cap D \cap F - C) \cup (A \cap B \cap F - E) \cup (A \cap F - C - E)$. Using rule 1 removes the product $A \cap B \cap D \cap F$ as B and D don't intersect. Rule 2 will reduce the products $A \cap D \cap F - C$ and $A \cap B \cap F - E$ to $A \cap D \cap F$ and $A \cap B \cap F$ as the complemented primitives do not intersect the product. Rule 4 removes the product $A \cap B \cap F$, rule 5 reduces $A \cap D \cap F$ to $A \cap D$ and rule 6 reduces $A \cap F - C - E$ to $A \cap F - E$. The normalized and geometric pruned CSG tree is then $(A \cap D \cap F) \cup (A \cap F - E)$. Expanding to partial products gives $(A_f \cap D \cap F) \cup (D_f \cap A \cap F) \cup (F_f \cap A \cap D) \cup (A_f \cap F - E) \cup (F_f \cap A - E) \cup (E_b \cap A \cap F)$. Finally, difference pruning will reduce $E_b \cap A \cap F$ to $E_b \cap A$ (rule 7) and $F_f \cap A - E$ to $F_f \cap A$ (rule 8).

We also prune products against the viewing volume for the current frame and classify trimming primitive bounding boxes against the near clipping plane to determine whether the extra capping step is necessary.

5. Interactive Rendering

We have incorporated our rendering algorithm in a simple, interactive solid modelling system built with standard components. The main framework is provided by the Inventor object-oriented 3D toolkit [15]. A model is represented by a directed acyclic graph of *nodes*. Operations on models, such as rendering or picking, are performed by means of *actions*. The toolkit may be extended by providing user written nodes and actions. Conventional solid modelling operations are provided by the ACIS geometric modeller [16]. ACIS is an object-oriented, boundary representation, solid modelling kernel.

Our modelling system adds new node types to Inventor which support ACIS modelled solids and CSG trees of solids. We also add a new rendering action which uses our stencil buffer CSG display algorithm to render CSG trees described by Inventor node graphs. A CSG evaluate action uses ACIS to fully evaluate a CSG tree allowing the tree to be replaced with a single evaluated solid node. All the standard Inventor interactive tools are available for editing models.

The system supports large CSG trees while maintaining interactive rendering speeds. During display and editing of a large CSG tree, only a small part of the model will be changing at any time. We "cache"

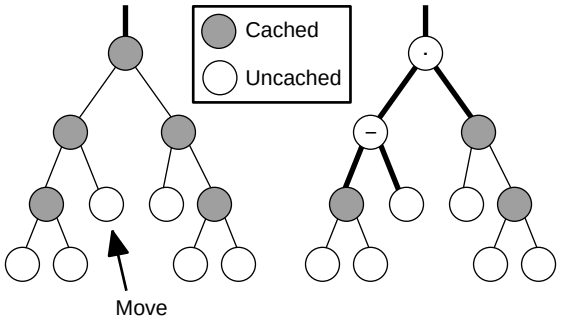

Figure 7: *Direct rendering of a CSG tree with cached geometry : (a) all caches valid, (b) limited direct rendering when a primitive is moved*

fully evaluated geometry obtained from the solid modeller at each internal node in the CSG tree. As caches become invalidated through editing of the model, portions of the tree are rendered directly (see figure 7), while the cached geometry is re-evaluated in the background (possibly on other workstations in a common network).

Current use of the system follows a common pattern. A user will quickly position and combine primitives using the solid modelling capabilities. During this stage the model is simple enough for the user to envisage the CSG operations required and to position primitives correctly. Figure 8 shows an example model of two intersecting corridors. Firstly, the space occupied by the corridors is modelled using 5 cubes and two cylinders which are unioned together. The corridors are then subtracted from a block. At this point the user wanted to position a skylight through the intersection of the corridors. Unsure of the exact positioning required, or the sort of results possible, the user roughly positioned a cylinder (the hole) and subtracted it from the model. A transparent instance of the primitive is also displayed by the system for reference. A manipulator was then used to drag the hole through the model revealing an unexpected new form. When satisfied with the positioning the hole is "fixed" in position. The fixing process doesn't change the internal representation of the model (it's still a complete CSG tree). It merely hides the apparatus used for interactive manipulation of the hole. The hole can be unfixed at any time and repositioned. This process of rough positioning, boolean combination and precise editing is then repeated.

6. Performance

The time complexity of our algorithm is proportional to the number of rendering operations carried out. We shall consider the rendering of one surface as a single rendering operation. Each pixel oriented "bookkeeping" operation is considered as an equivalent single unit. These operations have a lower geometry overhead than surface rendering but access more pixels. Equivalent functionality could be achieved by performing the bookkeeping operations with a repeated surface render. As in [13], we ignore the negligible normalization and pruning cost. We present the results for our current implementation of the algorithm. For reasons of clarity, some operations are described separately in section 3, whilst being implemented as a single operation.

Table 3 shows the number of rendering operations required for simple steps within the algorithm. The rendering algorithm is $O((kj)^2)$ for each product where j is the number of primitives in the product and k is the convexity of the primitives. The number of products generated by tree normalization is dependent on the structure of the tree and the geometry of the primitives with a worst case exponential relationship between number of primitives and products. In practice, both we, and Goldfeather et. al. [13], have found that the number of products after pruning is between $O(n)$ and $O(n^2)$ in the total number of primitives. The average product length, j, tends to be small and independent of the total number of primitives. Where long products arise they tend to be of the form $A-B-C-D-E...$ and are susceptible to difference pruning.

Table 4 provides performance statistics for the eight sample models in figure 9. The images are 500 by 500 pixels and were rendered on a Silicon Graphics 5 span 310/VGXT with a single 33Mhz R3000 processor. The VGXT has an 8 bit stencil buffer. The first part of the table provides statistics on normalization and pruning. We include the number of primitives in the CSG expression, total triangles used to represent the primitives and the number of passes required. The number and average length of partial products produced by normalization with and without pruning are given. The second part of the table provides a breakdown of rendering operations into target rendering, classification & trimming and bookkeeping operations. The third part of the table provides a breakdown of rendering time in seconds; both for rendering operations and depth buffer save/restore time. The depth buffer save/restore time is given for the general case algorithm and for the optimization possible when the model is the first thing rendered in the current frame.

Table 5 shows rendering times together with number of passes required for different stencil buffer sizes. The increases in time are modest because the implementation only saves and restores the areas of the depth

Convexity	1-convex ($k=1$)		k-convex	
Clipping	None	Near	None	Near
Classify Target Surface	k	k	$k+1$	$k+1$
Trimming Primitive	$2k+1$	$4k+1$	$2k+1$	$4k+1$
Render Target Surface	k	k	$k+1$	$k+1$

Table 3: *Rendering Operations per Step*

Model	a	b	c	d	e	f	g	h(part)	h(full)
Primitives	2	4	7	31	4	8	2	12	72
Triangles	96	256	408	1532	176	496	1928	8888	5536
Partial Products	2	6	32	34	5	14	3	13	72
Average Length	2	3	4	20.4	2.6	7	2	1.2	2.6
Partial Products (pruned)	2	6	32	34	5	14	3	13	72
Average Length (pruned)	2	3	4	2.7	2.6	3	2	1.2	2.3
Passes	1	1	1	5	1	2	1	1	11
Target Render Ops	2	4	7	30	4	8	5	25	72
Classification & Trimming Ops	4	18	128	92	13	42	8	8	164
Bookkeeping Render Ops	4	18	128	92	13	42	10	10	164
Total Render Ops	10	40	263	214	30	92	23	43	400
Target Time	0.005	0.003	0.038	0.039	0.008	0.017	0.031	0.009	0.118
Classification & Trimming Time	0.026	0.049	0.405	0.268	0.052	0.104	0.033	0.011	0.178
Bookkeeping Time	0.023	0.056	0.180	0.197	0.024	0.108	0.016	0.078	0.098
Save and Restore Time (general)	0.103	0.100	0.114	0.239	0.088	0.217	0.039	0.009	0.236
Save and Restore Time (first)	0.004	0.004	0.001	0.136	0.001	0.111	0.002	0.000	0.214
Total Time (general)	0.165	0.215	0.668	0.772	0.182	0.434	0.126	0.075	0.673
Total Time (first)	0.066	0.119	0.555	0.669	0.095	0.328	0.089	0.067	0.650

Table 4: *Rendering times (seconds) and statistics*

Stencil Bits	8	7	6	5	4	3	2
Model (c)	0.668(1)	0.670(2)	0.701(2)	0.735(2)	0.763(3)	0.782(4)	0.801(7)
Model (d)	0.772(5)	0.779(5)	0.804(6)	0.818(8)	0.837(10)	0.866(15)	0.884(30)
Model (f)	0.434(2)	0.435(2)	0.442(2)	0.426(2)	0.449(3)	0.475(4)	0.504(8)

Table 5: *Rendering time and number of passes with varying stencil size*

buffer that are changed during the classification stage. If less work is done in each pass the changed depth buffer areas typically become smaller. There is scope for further optimization of save and restore as the variations in times for the same number of passes shows. The different stencil buffer size causes a change in the composition of product groups. Placing partial products whose projected bounding boxes overlap into the same product groups will reduce the total area to be saved and restored.

Our algorithm performs particularly well in the sort of situations encountered within our interactive modelling system. Typically there is only ever one "dynamically" rendered CSG expression, usually involving a simple 1-convex "tool" and a more complex "workpiece" (figure 9(g)). Often we can achieve better performance by ignoring the top most caches of complex workpieces in order to expose more of the CSG tree to pruning. For example, in figure 9(h) an expression like $(A \cup B \cup C \cup D \cup ..) - X$ can be pruned to $A - X \cup B \cup C \cup D \cup ...$ This can vastly reduce both the number of polygons to be rendered (about 3–5 times as many polygons have to be rendered for $A - B$ compared to $A \cup B$) and the size of the screen area involved in bookkeeping and depth buffer save and restore operations. We provide rendering times for both cached (table 4 h(full)) and uncached cases (table 4 h(part)) of figure 9(h). The coloured primitives are those that are being "moved", the other geometry can be rendered from caches. The version that makes use of the caches is about 9 times faster than the fully rendered version. However, the triangle count is higher because the cached geometry has a more complex boundary than the original primitives.

Our implementation's performance compares well with that obtained by specialized hardware and pure software solutions. Figure 9(d) is our version of a model rendered by Goldfeather et. al. [13] on Pixel-Planes 4. They report a total rendering time of 4.02 seconds compared with our time of 0.67 seconds. The VGX architecture machine used for our tests was introduced in 1990 when Pixel-Planes 4 was nearing the end of its lifetime. Pixel-Planes 5 (the most recent machine in the Pixel-Planes series [17]) has performance some 50 times better than Pixel-Planes 4 on a full system with 32 geometry processors and 16 renderers. Such a system would have performance 10 times that of our implementation — at a far greater cost.

Figure 9(f) is our version of a model rendered by Thibault and Naylor's BSP tree based algorithm [6]. Their total rendering time is 7.2 seconds for a model with 158 polygons on a VAX 8650. Our time is 0.3 seconds for a model with 496 triangles. Our algorithm also scales better with increasing numbers of polygons ($O(kn)$ compared with $O(nlogn)$).

6.1. Other implementations

We have also implemented the algorithm using OpenGL [2] and tested it on our VGXT, a Silicon Graphics R3000 Indigo with starter graphics, and an Indigo^2 Extreme. The algorithm should run under any OpenGL implementation. On the systems we tested performance was comparable to the GL version in all areas except depth buffer save and restore. This operation was about 100 times slower than the GL equivalent. The problem appears to be a combination of poor performance tuning and a specification which requires conversion of the depth buffer values to and from normalized floating point. This problem should be resolved with the release of more mature OpenGL implementations. Single pass renders with the frame start optimization (the common case for our interactive modeller) run at full speed.

7. Conclusion

We have presented an algorithm which directly renders an arbitrary CSG tree and is suitable for use in interactive modelling applications. Unlike Goldfeather et. al. [13], our algorithm requires only a single color buffer, a single depth buffer, a stencil buffer and the ability to save and restore the contents of the depth buffer. It can be implemented on many graphics workstations using existing graphics libraries. Like Rossignac, Megahed and Schneider [14], the algorithm can display cross-sections of solids using clipping planes but is far more flexible. For instance, the algorithm could be used to directly display interferences between solids by rendering the intersection of the solids.

The algorithm has been implemented on an SGI 310/VGXT using the GL graphics library and has been integrated into an experimental modelling system. Performance compares well with specialized hardware and pure software algorithms for complete evaluation and rendering. The algorithm performs particularly well for incremental updates in an interactive modelling environment.

Acknowledgements

This work has been funded by Informatix Inc., Tokyo. Our thanks go to them for their support of the Martin Centre CADLAB over the last four years. Brian Logan, Paul Richens and Simon Schofield have all provided valuable insights and comments; as have the anonymous referees. Paul Richens created the models in figure 9 (g) and (h) using our interactive modeller.

References

1. P. McLendon, *Graphics Library Programming Guide*. Silicon Graphics, Inc., (1991). Document Number 007-1210-040.
2. OpenGL Architecture Review Board, *OpenGL Reference Manual*. Addison Wesley, (1992).
3. F. Jansen, "CSG hidden surface algorithms for vlsi hardware systems", in *Advances in Computer Graphics Hardware I*, Springer Verlag, (1987).
4. G. Kedem and J. L. Ellis, "The raycasting machine", in *Proceedings of the 1984 International Conference on Computer Design*, pp. 533–538, (October 1984).
5. J. Rossignac and J. Wu, "Depth-interval buffer for hardware-assisted shading from CSG: Accurate treatment of coincident faces and shadows", in *Fifth Eurographics Workshop on Graphics Hardware* (D. Grimsdale and A. Kaufman, eds.), (1989).
6. W. C. Thibault and B. F. Naylor, "Set operations on polyhedra using binary space partitioning trees", in *SIGGRAPH '87*, pp. 153–162, (1987).
7. S. D. Roth, "Ray casting for modeling solids", *Computer Graphics and Image Processing*, **18**(2), pp. 109–144 (1982).
8. P. R. Atherton, "A scanline hidden surface removal procedure for constructive solid geometry", in *SIGGRAPH '83*, vol. 17, pp. 73–82, (July 1983).
9. N. Okiro, Y. Kakuzu, and M. Marinoto, "Extended depth buffer algorithms for hidden-surface visualization", *IEEE Computer Graphics and Applications*, **4**(5), pp. 79–88 (1984).
10. J. R. Rossignac and A. A. G. Requicha, "Depth buffering display techniques for constructive solid geometry", *IEEE Computer Graphics and Applications*, **6**(9), pp. 29–39 (1986).
11. A. L. Thomas, "Geometric modelling and display primtives towards specialized hardware", in *SIGGRPAH '83*, vol. 17, pp. 299–310, (1983).
12. F. W. Jansen, "Depth-order point classification techniques for CSG display algorithms", *ACM Transactions on Graphics*, **10**(1), pp. 40–70 (1991).
13. J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs, "Near real-time CSG rendering using tree normalization and geometric pruning", *IEEE Computer Graphics and Applications*, **9**(3), pp. 20–28 (1989).
14. J. Rossignac, A. Megahed, and B.-O. Schneider, "Interactive inspection of solids: Cross-sections and interferences", in *SIGGRAPH '92*, vol. 26, pp. 353–360, (1992).
15. P. S. Strauss and R. Carey, "An object-oriented 3d graphics toolkit", in *SIGGRAPH '92*, vol. 26, pp. 341–349, (1992).
16. Spatial Technology inc., *ACIS Geometric Modeler Programmers Manual*, (September 1993).
17. H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel–planes 5: A heterogenous multiprocessor graphics system using processor–enhanced memories", in *SIGGRAPH '89*, vol. 23, pp. 79–88, (1989).

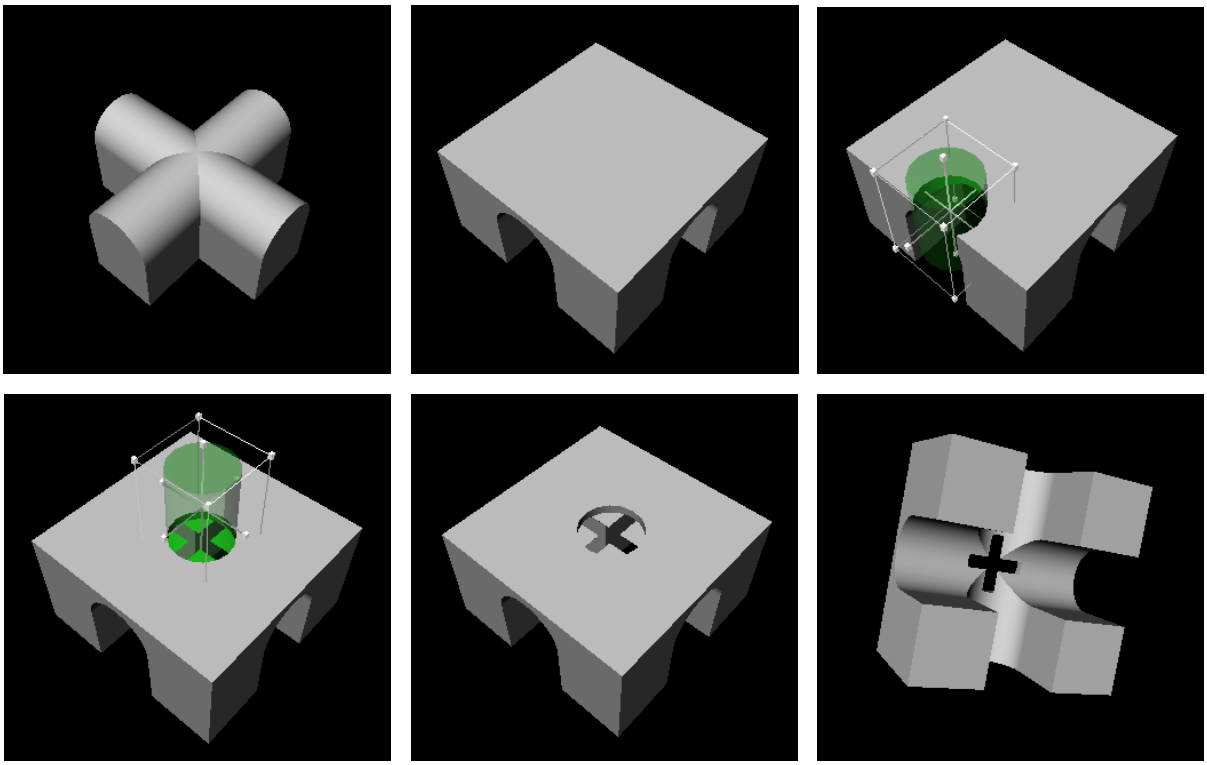

Figure 8: *Dragging a hole though the model reveals an unexpected new form*

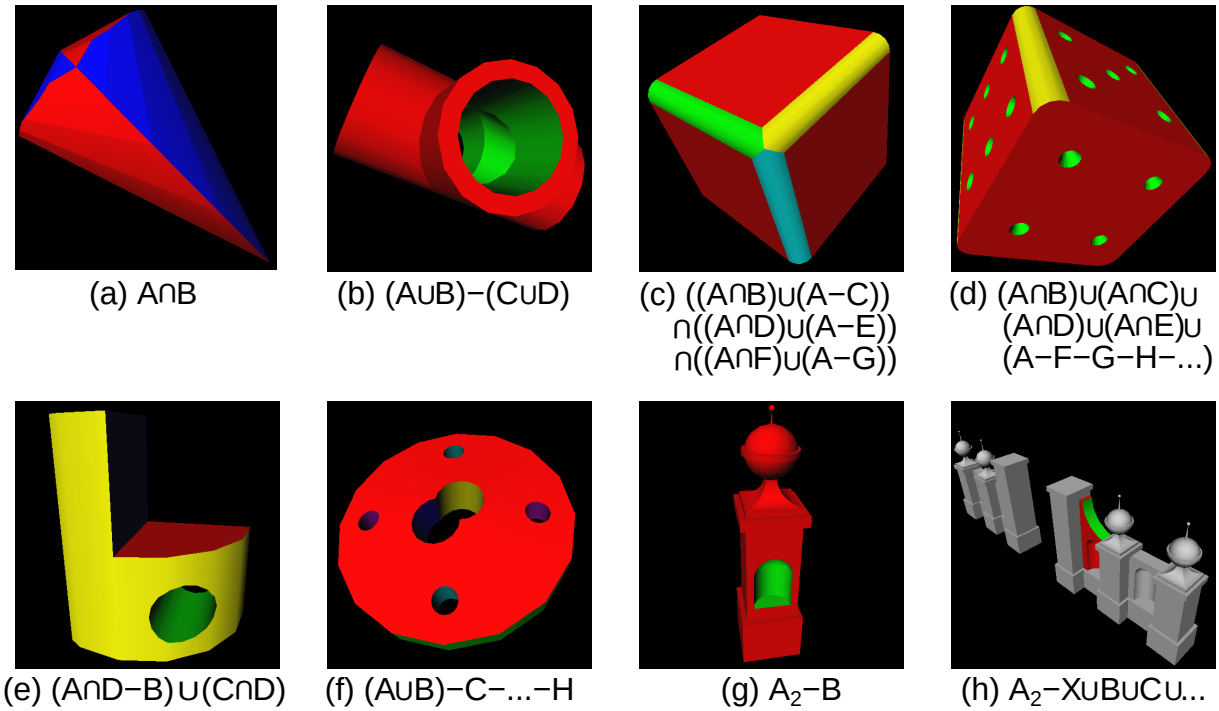

Figure 9: *Images generated by the stencil buffer CSG algorithm*