A Z-Buffer CSG Rendering Algorithm for Convex Objects

Nigel Stewart[1] **Geoff Leach**[2] **Sabu John**

Department of Mechanical and Manufacturing Engineering
RMIT University
Melbourne, Australia

ABSTRACT

A new algorithm for image-space CSG rendering is presented, based on subtraction of convex objects in a specific sequence. The algorithm has been implemented on OpenGL PC graphics hardware, as well as SGI workstations. Advantages of the algorithm include simpler implementation, closer affinity to hardware capabilities and comparable performance to other image-space CSG algorithms. The new algorithm is described, and compared to previous algorithms, experimentally and theoretically. Some graphics hardware issues related to image-space CSG performance are also discussed.

Keywords: Rendering Algorithms, Constructive Solid Geometry, OpenGL, Solid Modelling, Numerically Controlled Verification

1 INTRODUCTION

Constructive Solid Geometry[Requi80a] (CSG) is an approach to geometric modeling which applies Boolean operations volumetrically, according to relationships in a *CSG tree*. Parent nodes in a CSG tree represent operations such as Union ($\cup$), Intersection ($\cap$) and Difference (-). Leaf nodes of the tree represent primitive objects. It is common to associate transformations with each node, to allow flexible scaling, rotation and translation of components of the CSG model. Figure 1 illustrates a CSG model composed of a sphere, box and two cylinders.

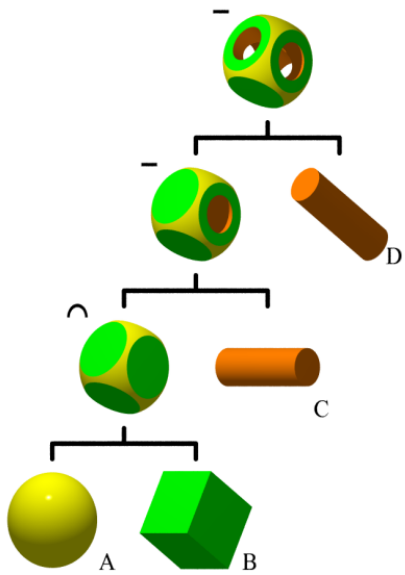


Figure 1: CSG Tree

The task of rendering an image of a CSG tree can be approached in different ways. *Object-space* approaches, such as boundary evaluation[Requi80a, Requi85a] can be used to convert from CSG to triangles, which are passed directly to a rendering pipeline such as OpenGL[OglArb93a, OglArb92a]. Alternatively, an algorithm rendering directly from the CSG tree can be utilised. *Image-space* approaches perform view-dependent surface clipping in addition to visible surface determination on a per-pixel basis. Image-space approaches include ray-casting, scanline algorithms

[1] nigels@nigels.com
[2] RMIT Department of Computer Science

and hardware-based z-buffer algorithms. We use the term *clipping* in a generic sense, referring to the process of determining which surface subset volumetrically satisfies the constraints of the CSG model. Visible surface determination is the process of finding the visible surface for each pixel.

This paper introduces a new image-space z-buffer CSG rendering algorithm, which we call *Sequenced Convex Subtraction*, or SCS. The algorithm is described in the context of previous approaches, such as the Goldfeather[Goldf86a] and Trickle[Epste89a] algorithms. OpenGL implementations of the SCS and Goldfeather algorithms have been developed to verify the SCS approach and to examine performance issues. Frame-rate results are presented for a range of CSG models on three different hardware platforms.

The primary advantage of the SCS algorithm is significantly less z-buffer copying, due to the reduced demands on multiple z-buffers for intermediate results. The algorithm is also simpler to implement. Other computer graphics problems make trade-offs between the generality of handling concave objects, and the simplicity and efficiency of computation on convex objects. OpenGL for example, rasterizes only convex polygons — concave polygons are tesselated into convex polygons by higher level software. Z-Buffer algorithms such as Goldfeather and Trickle perform conversion to convex geometry in image space. The SCS algorithm requires convex conversion in object space, rather than repeating this work for each frame.

2 CSG RENDERING ALGORITHMS

2.1 The Goldfeather Algorithm

The Goldfeather CSG rendering algorithm [Goldf86a, Goldf89a] is based on clipping one primitive in the z-buffer at a time. A second z-buffer is required to accumulate the z-buffer result, in the usual z-less test manner. The algorithm assumes that the CSG tree is converted to *sum-of-products* form. It has been shown[Goldf86a, Goldf89a] that any CSG tree may be *normalised* into sum-of-products form: $P_1 \cup P_2 \cup ... \cup P_p$, where p is the number of products. A product consists of objects related by only intersection and difference operations.

The Goldfeather Algorithm is outlined below. The near and far clipping planes of the viewing system are denoted as Z_{near} and Z_{far}.

```
Goldfeather:
  initialise output z-buffer to Z_far
  for each product P in CSG tree
    for each primitive Q in P
      if Q is subtracted
        draw back of Q into temp z-buffer
      if Q is intersected
        draw front of Q into temp z-buffer
      for each primitive R in P
        if R is not Q
          clip temp z-buffer against R
      merge temp z-buffer into output z-buffer
```

The operation of the Goldfeather algorithm is illustrated in Figure 2, using the CSG tree from Figure 1. Each row corresponds to a pass of the algorithm. The first column shows the primitive selected for clipping in the current pass. In the second column, the appropriate front or back surface is drawn into the z-buffer. In the third column, the z-buffer surface is clipped against all other primitives in the product. The final step is to merge the clipped z-buffer into the output z-buffer.

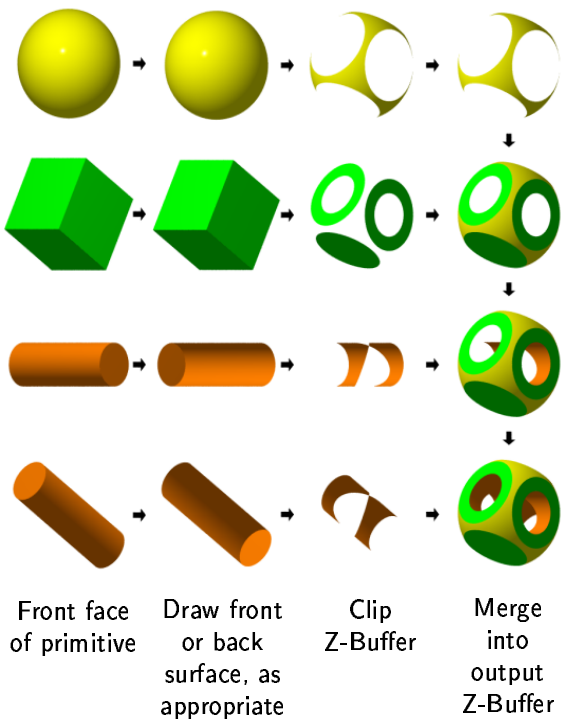


Figure 2: Goldfeather CSG Rendering Algorithm

Clipping of a z-buffer is implemented by counting the number of surfaces in front of each pixel. An

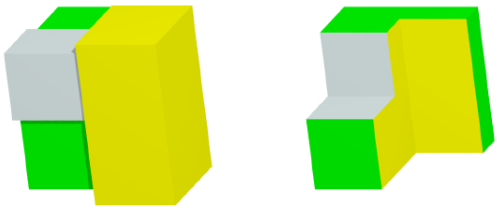


Figure 3: CSG Object: X - A - B

odd number means that the pixel is inside the volume of the primitive. This can be implemented in OpenGL by toggling a stencil bit[OglArb93a] at each pixel for every rasterised surface closer than the current one.

Concave primitives are handled by clipping each *layer* of the primitive in a separate pass. The number of passes corresponds to the depth complexity of the object, since only one z-buffer surface can be clipped at each pixel per pass. A simple algorithm can be written in OpenGL to draw the n'th layer of a concave primitive[Goldf89a, Stewa98a]. The stencil buffer is used to increment a counter for each fragment drawn into a pixel, updating the z-buffer only when the counter equals the desired layer.

The Goldfeather algorithm has been implemented in OpenGL[Stewa98a, Wiega96a]. Two z-buffers are required by the algorithm — the additional z-buffer can be simulated by copying between the z-buffer and system memory. The basic requirements are therefore a single colour buffer, a single z-buffer, a stencil buffer and the ability to save and restore the contents of the z-buffer[Wiega96a].

Z-buffer copying can be a significant bottleneck on OpenGL hardware platforms [Stewa98a, Wiega96a]. The *Layered Goldfeather Algorithm*[Stewa98a] can reduce this problem in certain circumstances by clipping layers, rather than primitives. The idea is to take advantage of depth complexity k, resulting in $O(kn)$ time, rather than $O(n^2)$. Major speedups are observed when $k < n$. Object-space optimisations have also been proposed [Goldf89a, Wiega96a].

2.2 The Trickle Algorithm

The Trickle CSG rendering algorithm[Epste89a] subtracts layers from front to back with respect to the viewing direction. Two auxillary z-buffers are used to iterate through the sequence of layers, the accumulated result is stored in a z-buffer, and another z-buffer is used as a temporary scratch-space — a total of four z-buffers. The Trickle algorithm is designed to handle CSG products in the form: $X - O_1 - ... - O_p \cap O_{p+1} \cap ... \cap O_n$. Although the algorithm operates on a subtractive basis, intersection can be supported via inversion.

Trickle:
- initialise output z-buffer to Z_{far}
- draw front surface of base primitive
- for each layer of subtracted volumes
  - extract the front surface of n'th layer into front z-buffer
  - extract the back surface of n'th layer into back z-buffer
  - replace output z-buffer with back z-buffer where frontZ < outputZ < backZ
- replace output z-buffer with Z_{far} where outputZ > back surface of base primitive

Figure 4 illustrates application of the Trickle algorithm to a CSG tree of three boxes ($X - A - B$) as illustrated in Figure 3. Initially, the front of the base object X is drawn into the z-buffer. The closest front-facing and back-facing surfaces of subtracted objects are determined for each pixel. This forms the first layer, which is subtracted from the z-buffer. Subtraction replaces the z-buffer with L_{back}, the back-facing surface of the layer, for each pixel where $L_{front} < z < L_{back}$. Then, the next layer is formed by finding the next closest front and back facing surfaces for each pixel. Each layer is subtracted from the z-buffer in turn, from nearest to furthest. Layers are view-dependent, since changing the viewing direction affects the relative distances of surfaces to the viewer.

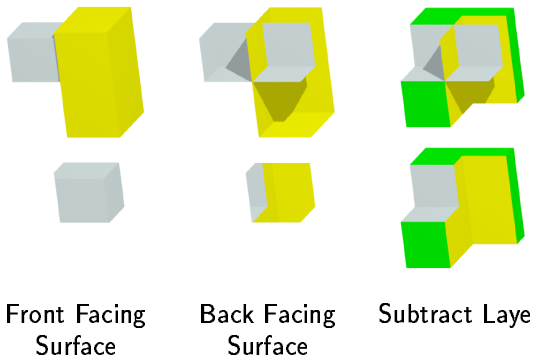


Figure 4: Trickle CSG Rendering Algorithm

The subtraction of an individual convex object (or layer) is illustrated in Figure 5. The Trickle algorithm repeats Steps (b) to (d) for each layer of subtracted volume.

The Trickle algorithm depends on layers being subtracted in front to back order. Since the z-

(a) Draw first surface into z-buffer

(b) Compare z-buffer to front and back surfaces

(c) Flag pixels inside volume

(d) Replace flagged pixels with back surface

(e) Reset regions of complete subtraction

Figure 5: Convex Subtraction From Z-Buffer

buffer has capacity for only one surface per pixel, it is not possible to account for holes behind the current z-buffer surface. By ensuring that closer subtractions are performed earlier, holes behind the current z-buffer surface are known to be occluded, and can be ignored.

Layers are extracted iteratively, based on the previous layer. The algorithm searches for the closest surface which is further than the current layer surface. Incoming triangles are depth tested against two different z-buffers. Fragments closer than the current layer z-buffer are rejected. Fragments further than the "closest seen so far" in the temporary z-buffer are also rejected. Once all rasterisation is complete, the temporary z-buffer contains the next closest surface and replaces the contents of the layer z-buffer. A collision count for each pixel is maintained, to handle the case where more than one surface occupies the same z value at the same pixel. The layer extraction algorithm is as follows:

```
ExtractNextFrontFacingLayer:
  on first pass
    initialise front z-buffer to Z_near
    initialise collision count to zero
  on subseqent passes
    decrement collision count for each pixel
  initialise temp z-buffer to Z_far
  for each front facing surface S
    replace temp z-buffer with S where
      frontZ < S < tempZ
    and
      collision count is zero
  for each pixel where collisions is zero
    count collisions
```

Extraction of the back-facing surface of the layer is performed similarly, except that back-facing surfaces are considered rather than front-facing.

It should be noted that the architecture of OpenGL does not support depth testing of a fragment against two z-buffers. This can be simulated in OpenGL, but is costly. Other architectures supporting deep frame-buffers, multiple z-buffers, extended fragment testing or high-bandwidth are more promising[Eyles97a, Molna88a, Molna92a, Rossi90a].

The Trickle algorithm handles intersections by subtracting inverted primitives. The inverse of a convex primitive X spanning $[X_{front}, X_{back}]$ is $[Z_{near}, X_{front}]$ and $[X_{back}, Z_{far}]$. Implementing this inversion involves swapping the front and back-facing surfaces, and "tricking" the layer extraction algorithm. Initialising the front collision count for each pixel to the number of intersected primitives simulates the Z_{near} surfaces. The necessary Z_{far} surfaces result from the convergence of the layer extraction algorithm to Z_{far}.

3 THE NEW ALGORITHM

3.1 SCS Overview

The *Sequenced Convex Subtraction* (SCS) CSG rendering algorithm is based on subtraction of convex objects in a specific sequence. The SCS algorithm operates on a similar basis to Trickle, in that convex volumes are subtracted from the z-buffer. However, rather than extracting a sorted sequence of layers, convex primitives are subtracted in a sequence that handles every possible order in the viewing direction. In the best case, no pixel is covered by more than one primitive, and any sequence that includes all primitives will suffice. In the worst case, all primitives overlap the same pixel and all $n!$ scenarios need to be catered for. Interestingly, these can all be encoded in an n^2 length sequence.

Figure 6 illustrates the subtraction sequence necessary to handle the two possible dependencies

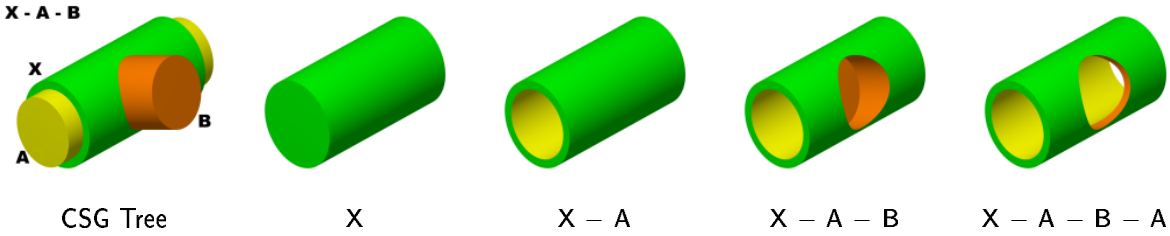


Figure 6: Subtraction sequence for two objects.

between two subtracted objects. While the sequence $X-B-A$ would be sufficient in this particular case, the sequence $X-A-B-A$ is assured to work for any viewing direction or position of the primitives. In fact, the same sequence will work for any CSG tree in the form: $X-O_1-O_2$.

The SCS CSG rendering algorithm can only handle convex objects. Concave objects must be decomposed into convex primitives in object-space. This idea is also reflected in the design of a hybrid object-space/image-space CSG rendering algorithm [Rappo97a].

SCS performs no image-space surface sorting, in contrast to the Trickle algorithm. It can be cheaper to perform redundant subtractions than to sort. SCS requires no z-buffer management for a single product, in contrast to both the Goldfeather and Trickle algorithms. However, the convex representation required by SCS results in more surface information than needs to be processed by other algorithms. The performance implications are examined in Section 4.3.

3.2 SCS Product

The SCS Algorithm for rendering a CSG product is as follows:

```
SCSProduct:
  initialise z-buffer to Z_far
  draw front surface of base primitive
  for each E in SCS sequence
    if primitive[E] is subtracted
      for each pixel in range of primitive[E]
        replace z-buffer with primitive[E].back
    if primitive[E] is intersected
      for each pixel closer than primitive[E].front
        replace z-buffer with primitive[E].front
      for each pixel further than primitive[E].back
        replace z-buffer with Z_far
      for each pixel not covered by primitive[E]
        replace z-buffer with Z_far
```

The first if-then block implements subtraction, as illustrated in Figure 5. The second if-then block implements intersection via logical inversion. The algorithm requires one z-buffer and no z-buffer copying.

3.3 SCS Tree

Rendering more than one CSG product requires some z-buffer management:

```
SCSTree:
  initialise output z-buffer to Z_far
  for each product P
    SCSProduct into temp z-buffer
    merge temp z-buffer into output
      with z-less test
```

This algorithm assumes that the CSG tree is in sum-of-products form, and merges products via the standard 'less than' z-buffer operation. It is similar to the outer loop of the Goldfeather algorithm, except that rather than merging clipped primitives, the SCS algorithm merges clipped products. There are usually fewer products in a normalised CSG tree than primitives.

3.4 Sequence Generation

The SCS algorithm performs CSG rendering by subtracting in a sequence that encodes every permutation of the objects. These sequences have been formulated abstractly as *permutation embedding sequences*[Galbi76a]. A sequence is embedded if it can be formed by deleting other entries. For example, *cab* is embedded in *abcbabc*: $\star\star c\star ab\star$, where $\star$ represents a deletion.

A simple algorithm can be used to construct permutation embedding sequences. The algorithm uses a permutation denoted s_1. Concatenation of

n copies of s_1 results in a sequence embedding every permutation. The length of these sequences is n^2. For example:

$n = 3$, $s_1 = abc$
$s_1 s_1 s_1 \rightarrow abc\ abc\ abc \rightarrow abcabcabc$

A shorter sequence can be obtained by alternating between s_1 and s_2, where s_2 is the reversal of s_1. At each boundary between s_1 and s_2 the repeated entries can be collapsed into one. For example:

$n = 3$, $s_1 = abc$, $s_2 = cba$
$s_1 s_2 s_1 \rightarrow abc\ cba\ abc \rightarrow abcbabc$

These can be proven to be permutation embedding sequences. Since each block contains each possible entry, any permutation can be formed by selecting the appropriate entry from each block. Collapsing repeated entries does not affect this property, since a different entry is required from neighbouring blocks.

Table 1 lists permutation embedding sequences for $1 \leq n \leq 6$. The length of these sequences is $n^2 - n + 1$. Shorter sequences[Galbi76a] can be generated for $n > 3$, but are more complicated. Determining optimal permutation embedding sequences is thought to be an open problem.

n	l	sequence
1	1	a
2	3	aba
3	7	$abcbabc$
4	13	$abcdcbabcdcba$
5	21	$abcdedcbabcdedcbabcde$
6	31	$abcdefedcbabcdefedcbabcdefedcba$

Table 1: Subtraction Sequences for $n \leq 6$

3.5 Depth Complexity Optimisation

If the maximum depth complexity of the subtracted objects is known, shorter subtraction sequences can be used. The sequences previously discussed cater for every permutation, but can be shortened to take depth complexity into account when $k < n$. k concatenated blocks are sufficient, where k is the maximum number of subtracted objects overlapping any pixel. For example:

$n = 3$, $k = 2$, $s_1 = abc$, $s_2 = cba$
$s_1 s_2 \rightarrow abc\ cba \rightarrow abcba$

The length of sequences in this form is $kn - k + 1$, or $O(kn)$.

Depth complexity can be determined with OpenGL by using the stencil buffer to count the number of objects covering each pixel[Stewa98a].

3.6 Limitations

Several general limitations apply to z-buffer CSG rendering algorithms, including the SCS algorithm. Primitive objects must have distinct front and back facing surfaces, and contain no holes or gaps in the tesselation. Also, the viewing system must be chosen so that no primitives span the near or far clipping planes.

4 IMPLEMENTATION

4.1 Overview

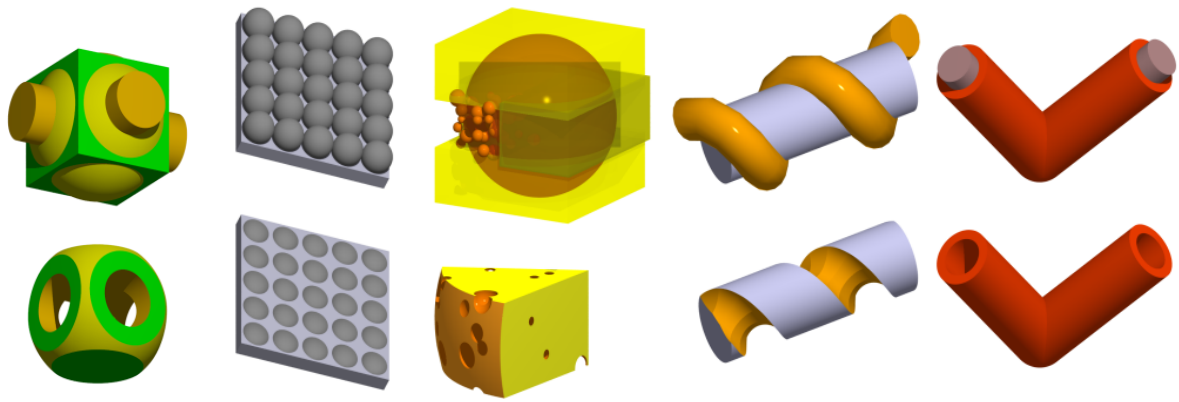
The Goldfeather[Goldf89a], Layered Goldfeather [Stewa98a] and the SCS CSG rendering algorithms have been implemented in C++ using OpenGL and GLUT. The implementation makes use of per-frame depth complexity sampling, but not object-space separability information. This presents a problem, since it is not known how to apply this information to SCS, but the Goldfeather algorithm makes effective use of it. This experiment can be regarded as an indication of CSG rendering integrated at a low level of the graphics environment, where this information could be expected to be absent. We intend to direct future efforts towards a clearer performance comparison between the SCS and Goldfeather approaches.

Three hardware OpenGL platforms have been used to obtain experimental results. The first is a Silicon Graphics O2, with a 180 Mhz MIPS R5000 processor, 160 Mbytes of RAM, running IRIX 6.3. The second is a Silicon Graphics Indigo2, with a 195Mhz MIPS R10000 processor, 128 Mbytes of RAM, High Impact graphics board running IRIX 6.2. The third is a 300Mhz Intel Pentium II, Windows NT 4.0 SP 5, and ASUS AGP TNT2 V3800 TVR 32MB Graphics Card. The OpenGL window is 200x200 pixels, 24 bit z-buffer and 8 bit stencil buffer.

4.2 CSG Objects

CSG objects used in this experiment are illustrated in Figure 7.

The *Widget* object is a CSG product with one intersection and two difference operations. The



(a) Widget (b) Grid (c) Swiss Cheese (d) Machine Tool (e) Hollow Pipe

Figure 7: CSG Test Cases.

depth complexity is constant from every viewing direction since all primitives overlap the center of the model.

The *Grid* object is a CSG product with 25 spheres subtracted from a rectangular block. Depth complexity varies according to the viewing direction.

The *Swiss Cheese* object is a large CSG tree involving subtraction of more than 50 convex objects. Despite the large number of objects, the depth complexity is often between 4 and 6.

The *Machine Tool* object consists of a tesselated helix subtracted from a cylinder. Two versions of the helical mesh have been used, a concave boundary representation for Goldfeather rendering, and a convex decomposition for SCS.

The *Hollow Pipe* dataset is three CSG products forming a bent hollow pipe.

4.3 Benchmark Results

The results indicate that the performance of SCS is comparable to the other algorithms tested. Rendering performance is sensitive to the model, viewing direction, and graphics hardware design.

The best overall speedup for SCS was observed on the TNT2 hardware. This environment has very fast triangle rasterisation, relative to z-buffer copying. The Indigo2 is the opposite, having relatively good bandwidth. The results for the machine tool model neatly illustrate the sensitivity of each algorithm to the hardware design.

As discussed previously, a more favorable comparison for the Goldfeather algorithm would include object-space separability information. In the absence of this information, SCS can achieve useful frame-rates for the models we investigated.

Dataset	Algorithm	Frame Rate (frame/sec)		
		O2	Indigo2	TNT2
Widget	Goldfeather	5.20	15.18	1.41
	Layered Gold.	3.51	10.85	1.01
	SCS Product	14.93	26.18	15.27
Grid	Goldfeather	0.31	0.73	0.15
	Layered Gold.	1.10	2.54	0.71
	SCS Product	1.64	3.66	2.67
Swiss Cheese	Goldfeather	0.09	0.34	0.06
	Layered Gold.	0.36	0.74	0.31
	SCS Product	0.46	0.98	0.74
Machine Tool	Goldfeather	5.41	18.98	1.46
	Layered Gold.	4.15	14.81	1.23
	SCS Product	4.93	11.12	6.95
Hollow Pipe	Goldfeather	1.48	4.74	0.41
	Layered Gold.	0.72	2.43	0.22
	SCS Tree	3.24	8.46	1.57

Table 2: Benchmark results.

5 CONCLUSION

The *Sequenced Convex Subtraction* (SCS) CSG rendering algorithm has been described, and shown to have certain advantages over previous algorithms. The basis of the algorithm is to subtract volumes in a sequence that caters for all possible orderings in viewing direction. Permutation embedding sequences have been shown to correctly render CSG products from different viewing directions.

Taking view dependent depth complexity into account, sequences of length $O(kn)$ produce the correctly rendered result. $O(kn)$ is asymptotically

similar to the performance of other algorithms. The advantage of the SCS Algorithm is fewer z-buffers and fewer z-buffer copying operations. Therefore, SCS is particularly advantageous in low-bandwidth environments.

5.1 FURTHER WORK

The SCS Algorithm is driven by a sequence encoding each possible depth dependency between subtracted volumes. View-specific depth complexity information results in an $O(kn)$ sequence. Additional information such as object-space separability can further reduce the sequence length. If two subtracted volumes are known not to intersect, there is no need to encode the dependency into the sequence. We would like to develop sequence generation techniques that make use of this information. It would be interesting to benchmark SCS and Goldfeather implementations making use of the same object-space separability information.

5.2 ACKNOWLEDGEMENTS

This work has been supported by the Co-Operative Research Center for Intelligent Manufacturing Systems & Technologies. Special thanks to Nik Lygeros and Robert Erra for their interest and contribution to the permutation embedding sequence problem. The POV-Ray 3.1 rendering package was used for many illustrations in this paper.

REFERENCES

[Epste89a] D. Epstein, F. Jansen, J. Rossignac, "Z-Buffer Rendering from CSG: The Trickle Algorithm", *IBM Research Report RC 15182*, Nov. 1989

[Eyles97a] 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

[Galbi76a] G. Galbiati, F. P. Preparata, "On Permutation-Embedding Sequences" *SIAM J. of Appl. Math.*, Vol. 30, No. 3, May 1976, pp. 421-423

[Goldf86a] 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

[Goldf89a] 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

[Molna88a] S. Molnar "Combining Z-buffer Engines for Higher-Speed Rendering" *Proc. Eurographics '88, Third Workshop on Graphics Hardware*, Sep. 1998, pp. 171-182

[Molna92a] S. Molnar, J. Eyles, J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition" *Siggraph 92*, pp. 231-240

[OglArb93a] OpenGL Architecture Review Board, *OpenGL Programming Guide*, Addison Wesley, California, 1993.

[OglArb92a] OpenGL Architecture Review Board, *OpenGL Reference Manual*, Addison Wesley, California, 1992.

[Rappo97a] A. Rappoport, S. Spitz, "Interactive Boolean Operations for Conceptual Design of 3-D Solids", *Siggraph 97*, pp. 269-278

[Requi80a] A. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems", *Computing Surveys*, Vol. 12, No. 4, Dec. 1980, pp. 437-464

[Requi85a] 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

[Rossi90a] 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

[Stewa98a] N. Stewart, G. Leach, S. John, "An Improved Z-Buffer CSG Rendering Algorithm", *1998 Eurographics/Siggraph Workshop on Graphics Hardware*, Aug 1998, pp. 25-30

[Wiega96a] T. F. Wiegand "Interactive Rendering of CSG Models" *Computer Graphics Forum*, Vol. 15, No. 4, Oct. 1996, pp. 249-261