Linear-time CSG Rendering of Intersected Convex Objects

Nigel Stewart, Geoff Leach and Sabu John

RMIT School of Computer Science and Information Technology
GPO Box 2476v
Melbourne, 3001
Victoria, Australia
nigels@nigels.com, gl@cs.rmit.edu.au

RMIT Department of Mechanical and Manufacturing Engineering
Sabu.John@rmit.edu.au

ABSTRACT

The *Sequenced Convex Subtraction* (SCS) algorithm is a hardware based multi-pass image-space algorithm for general purpose *Constructive Solid Geometry* (CSG) Rendering. Convex objects combined by volumetric intersection, difference and union are rendered in real-time without b-rep pre-processing. OpenGL stencil and depth testing is used to determine the visible surface for each pixel on the screen.
This paper introduces a specialised algorithm for CSG Rendering of intersected convex objects, we call SCS-Intersect. This new technique requires linear time with respect to the number of intersections. SCS-Intersect is primarily of interest as an optimisation to the SCS algorithm for rendering CSG trees of convex objects. A revised formulation of the SCS CSG Rendering algorithm is presented in this paper.

Keywords: CSG Rendering, Rendering Algorithms, Constructive Solid Geometry, OpenGL, Solid Modelling, Numerical Control (NC) Verification.

1 INTRODUCTION

As the performance of graphics hardware improves, multi-pass rendering techniques for anti-aliasing, shadowing, specular light, reflection and other effects become increasingly attractive for enhancing the visual realism of real-time interactive graphics. Over time, the graphics hardware feature-set has evolved to include stencil testing and programmability at the vertex and fragment level. This increased flexibility and performance enables the use of sophisticated graphics-hardware based algorithms for interactive applications.

The pervasive availability of z-buffer hardware for hidden surface elimination has encouraged the development of alternative applications of z-buffer graphics hardware. Applications include image compositing, shadow maps, voxelisation, discrete Voronoi diagrams, object reconstruction, symmetry detection and *Constructive Solid Geometry* (CSG) rendering[Theoh01].

Image-space CSG rendering techniques provide an alternative to object-space evaluation[Requi85] of CSG trees for geometric design or machining simulation. The relative simplicity and robustness of implementation, the interactive flexibility of dynamic CSG trees, and the potential for pixel parallelisation[Molna88] make image-space algorithms attractive for some applications.

Existing CSG rendering methods[Goldf89, Wiega96, Rappo97, Stewa00] generally require $O(n^2)$ time, where n is the number of leaf nodes in the CSG tree. This $O(n^2)$ requirement is due to the general strategy of comparing every pair of objects. This can be improved by taking advantage of depth complexity[Stewa98] (the number of objects covering each pixel) — resulting in $O(kn)$ execution time, where k is the maximum depth complexity. The *Sequenced Convex Subtraction* (SCS) algorithm[Stewa00] aims to improve CSG performance by reducing the requirement for z-buffer copying — a common bottleneck in contemporary graph-

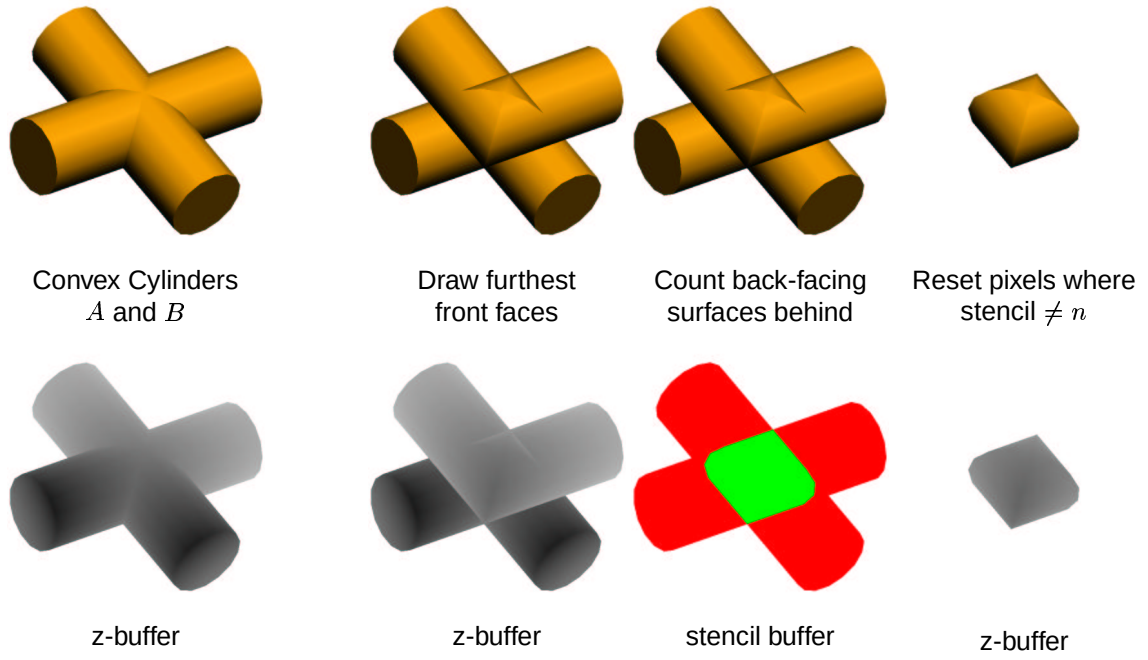


Figure 1: Z-Buffer Intersection using the z and stencil buffers

ics hardware[Wiega96, Stewa98, Stewa00].

This paper introduces a new approach for rendering the intersection of convex objects in linear time, that we call SCS-Intersect. It has been implemented using the z and stencil testing functionality of OpenGL[Boarda, Boardb]. SCS-Intersect can be used as a stand-alone algorithm for CSG trees of convex objects in the form: $A \cap B \cap C \cap ....$ It can also be utilised in a more general-purpose CSG rendering algorithm such as SCS[Stewa00].

The SCS-Intersect algorithm is described in Section 2, including some timing results for NVIDIA GeForce3 hardware. Section 3 refines the SCS CSG rendering algorithm, incorporating SCS-Intersect. Section 4 presents some SCS timing results for CSG trees including intersection and subtraction, followed by a conclusion and future work in Section 5.

2 SCS-INTERSECT

The intersection of two objects $A \cap B$ is the volume contained by both A and B. In terms of surfaces, the intersection is surfaces of A inside B and surfaces of B inside A. The SCS-Intersect algorithm uses OpenGL to render the intersection of any number of convex objects: $A \cap B \cap C \cdots$

Algorithms limited to convex objects often have advantages in terms of robustness and performance. Graphics hardware typically rasterises only triangles, and relies on layers of software for triangle tessellations of convex and concave polygons. Similarly, the SCS algorithms assume CSG trees of convex objects, and rely on the application to perform convex decomposition where necessary[Stewa00].

Convex objects characteristically have two distinct surfaces for each pixel - a “near” front-facing surface, and a “far” back-facing surface. The OpenGL backface culling mechanism provides a convenient mechanism for restricting rasterisation to either front or back-facing polygons. It also ensures that with culling enabled, there is at most one fragment per pixel, per object. The number of fragments processed at a pixel therefore corresponds to the number of objects covering that pixel.

Using OpenGL, the closest visible surface of the boolean intersection of a set of convex objects is formed in the z-buffer. An additional pass is performed with a z-equal z-test to determine the colour of each pixel.

2.1 Algorithm

The image-space intersection algorithm uses two principles. First, only the furthest front-facing surface can be volumetrically inside all of the objects. Closer front-facing surfaces cannot be volumetrically inside more distant objects. Second, the intersection

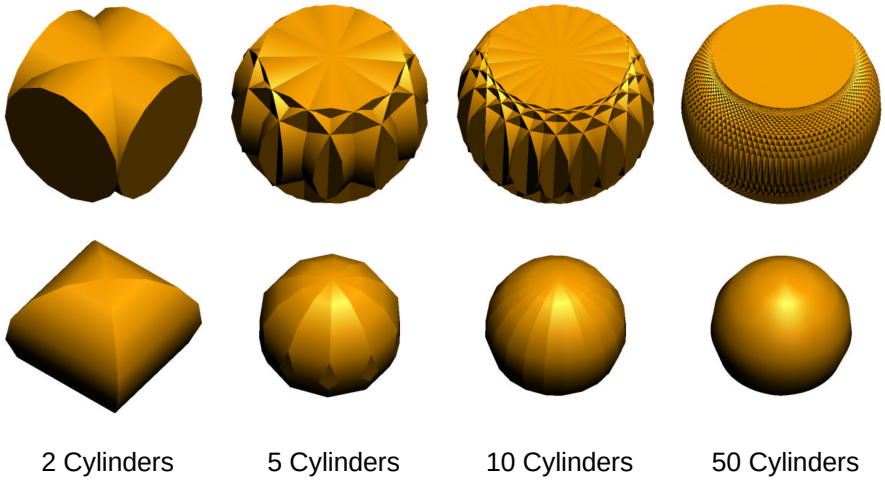


Figure 2: Application of SCS-Intersect to Cylinders

surface must be in front of n back-facing surfaces. If any back-facing surface is closer than the candidate z-buffer pixel, then the pixel can't possibly be inside the intersection. If the depth-complexity of a pixel is less than n, then it can't possibly be inside all n objects.

These two ideas are implemented as a multi-pass algorithm with appropriately configured z and stencil tests.

SCS-Intersect:

$n \leftarrow$ number of intersected objects
for each pixel
  $z \leftarrow Z_{near}$
  $stencil \leftarrow 0$
for each intersected object, i
  if $i_{front} > z$
    $z \leftarrow i_{front}$
for each intersected object, i
  if $i_{back} > z$
    $stencil \leftarrow stencil + 1$
for each pixel
  if $stencil \neq n$
    $z \leftarrow Z_{far}$
  $stencil \leftarrow 0$

To begin with, the z-buffer is initialised to Z_{near}, and the stencil buffer reset to zero. Then the furthest front facing surface is drawn into the z-buffer by rasterising with a z-greater test. The stencil buffer is then used to count the number of back-facing objects behind the z-buffer. Finally, all pixels that do not have n back-facing surfaces behind the z-buffer are reset to Z_{far}.

The intersection of two cylinders is illustrated in Figure 1. In the z-buffer diagrams, black is near and white is far. In the stencil-buffer diagrams white, red and green denote values of 0, 1 and 2 respectively.

2.2 Limitations

A maximum of $2^s - 1$ surfaces can be counted with an s-bit stencil buffer. On hardware with an 8-bit stencil buffer, the algorithm is therefore limited to 255 intersected objects. However, the last two passes could be incorporated into a loop to check $2^s - 1$ surfaces at a time, at the expense of extra passes. We expect that a limitation of 255 surfaces for an 8-bit stencil is reasonable for most practical purposes.

2.3 OpenGL Implementation

The following C code fragment implements the SCS-Intersect algorithm in OpenGL. Figure 2 illustrates the result of intersecting increasing numbers of cylinders.

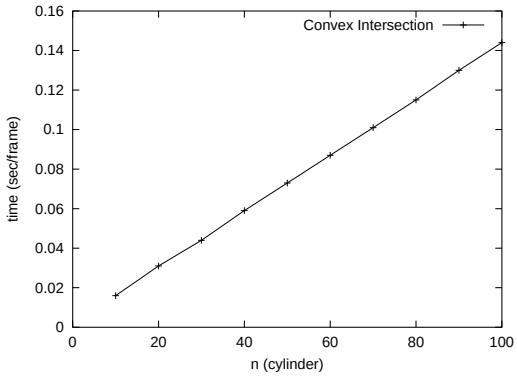
```
/* Clear frame-buffer */

glClearDepth(0.0);
glDepthMask(GL_TRUE);
glClearStencil(0);
glStencilMask(~0);
glColorMask(GL_TRUE,GL_TRUE,
            GL_TRUE,GL_TRUE);
glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);

/* Draw furthest front face */
```

```
glColorMask(GL_FALSE,GL_FALSE,
            GL_FALSE,GL_FALSE);
glDisable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_GREATER);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
drawCylinders();

/* Count back-facing surfaces behind */

glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS,0,~0);
glStencilOp(GL_KEEP,GL_KEEP,GL_INCR);
glDepthMask(GL_FALSE);
glCullFace(GL_FRONT);
drawCylinders();

/* Reset pixels where n != stencil */

glStencilFunc(GL_NOTEQUAL,n,~0);
glStencilOp(GL_ZERO,GL_ZERO,GL_ZERO);
glDepthFunc(GL_ALWAYS);
glDepthMask(GL_TRUE);
glDisable(GL_CULL_FACE);
drawZfar();

/* Draw RGB image */

glColorMask(GL_TRUE,GL_TRUE,
            GL_TRUE,GL_TRUE);
glDisable(GL_STENCIL_TEST);
glDepthFunc(GL_EQUAL);
glDepthMask(GL_FALSE);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
drawCylinders();
```

2.4 Timing Results

Timing results in this paper were obtained on a 1Ghz Pentium III system with NVIDIA GeForce3 OpenGL graphics, running RedHat Linux 7.1. The algorithms were implemented using C++, OpenGL and GLUT[Kilga96]. Images were rendered in an 1024x768 window, each gluCylinder having 15 slices and 10 stacks.

The performance of SCS-Intersect over a 1000 frame period is recorded in Table 1. The graph shows the linear relationship between the rendering time per frame, and the number of intersected cylinders.



n	Frame Rate (frame/sec)	Time (sec/frame)
10	59	0.016
20	33	0.031
30	22	0.044
40	17	0.059
50	14	0.073
60	11	0.087
70	10	0.101
80	9	0.115
90	8	0.130
100	7	0.144

Table 1: CSG-Intersect performance.

3 SCS ALGORITHM

The *Sequenced Convex Subtraction* (SCS) CSG rendering algorithm draws a CSG tree of convex objects using OpenGL graphics hardware. The algorithm operates in three phases: pre-processing, z-buffer processing, and final RGB image composition.

In the pre-processing phase, the CSG tree is converted to sum-of-products form by means of *tree normalisation*[Goldf89, Rossi94]. Each product consists of only intersection and subtraction operations, and the products are combined via union operations. The CSG tree $(A \cap B) \cup (C - D)$ consists of two products: $A \cap B$ and $C - D$. Each product is processed independently in the z-buffer processing phase, and merged into the final result using the z-less z-buffer test. Tree normalisation is view independent and only needs to be performed when the CSG tree changes. Refer to Goldfeather's papers[Goldf86, Goldf89] for further explanation of tree normalisation and the algorithm.

A second pre-processing step is to determine subtraction sequences for each product. The optimal subtraction sequence is in front to back order. However, it is sufficient to subtract in any sequence that embeds the correct subtraction sequence. It can be faster and generally much simpler to perform extra subtractions than to determine the ideal subtraction sequence for each viewing direction. Subtraction sequences can be determined either view-dependently or view-independently using the SCS-Sequence algorithm.

In the z-buffer processing phase the z-buffer result of each product is determined, and merged into a final z-buffer result. To begin with, intersected objects are handled using the SCS-Intersect algorithm. Then the subtraction sequence is used by the SCS-Subtract algorithm to process the subtracted objects. The SCS-ZClip algorithm clips the z-buffer against intersected objects, and completely subtracted pixels

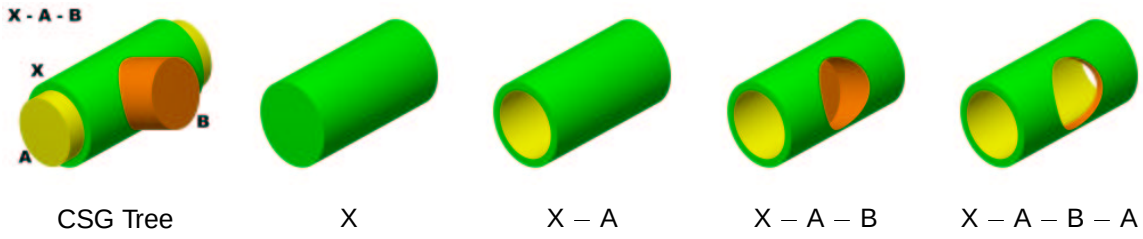


Figure 3: Subtraction sequence for two objects.

reset to Z_{far}. The product z-buffer is then merged into the final z-buffer with a z-less depth test.

In the final phase the merged z-buffer result is used for calculating the RGB image of the CSG tree. The front-facing surfaces of intersected objects, and the back-facing surfaces of subtracted objects are drawn with lighting enabled and a z-equal depth test.

In the following sections, the SCS-Sequence, SCS-Subtract and SCS-ZClip algorithms are discussed, followed by the full SCS algorithm.

The SCS CSG rendering algorithm as presented here includes revisions with respect to our previous presentation[Stewa00]. Intersected objects are now handled by the $O(n)$ SCS-Intersect algorithm, rather than by the $O(n^2)$ subtraction approach. In this version, the subtraction sequence need only include subtracted objects. The resulting performance improvement is verified experimentally in Section 4.

3.1 SCS-Sequence

Subtraction sequences must handle every possible dependency between subtracted objects in the product. A may reveal B, which in turn reveals C, which will only be rendered properly if A, B and C are subtracted in the right order. Figure 3 illustrates a subtraction sequence incorporating $-A-B$ and $-B-A$. For any viewing direction or configuration of two subtracted objects, the sequence $X-A-B-A$ ensures that both possible orderings are correctly handled.

Permutation embedding sequences[Galbi76] have the property that all $n!$ permutations of n objects are embedded. A sequence is embedded if it can be formed by deleting other entries. For example, CAB is embedded in $ABCBABC$: $\star\star C\star AB\star$. Permutation embedding sequences of length $O(n^2)$ or $O(kn)$ can be easily obtained[Stewa00].

Sequence encoding uses a permutation denoted s_1, and it's reversal s_2. The sequence is formed by alternating between s_1 and s_2 until n copies have been catenated. At each boundary between s_1 and s_2 the repeated entries are collapsed into one. The length of these sequences is $n^2 - n + 1$.

For example:

$n = 2$, $s_1 = AB$, $s_2 = BA$
$s_1 s_2 \rightarrow AB\ BA \rightarrow ABA$

$n = 3$, $s_1 = ABC$, $s_2 = CBA$
$s_1 s_2 s_1 \rightarrow ABC\ CBA\ ABC \rightarrow ABCBABC$

If the maximum depth complexity of the subtracted objects is known, shorter subtraction sequences of $O(kn)$ length can be used, where k is the maximum number of subtracted objects overlapping any pixel. The depth complexity can be found easily using a stencil test[Stewa00].

For example:

$n = 2$, $k = 1$, $s_1 = AB$, $s_2 = BA$
$s_1 \rightarrow AB$

$n = 3$, $k = 2$, $s_1 = ABC$, $s_2 = CBA$
$s_1 s_2 \rightarrow ABC\ CBA \rightarrow ABCBA$

3.2 SCS-Subtract

Subtracted objects in a CSG product are handled by sequenced subtraction[Stewa00] from the z-buffer. Each subtraction involves comparing the front and back facing surfaces to the z-buffer. The z-buffer is updated for each pixel volumetrically inside the subtracted object.

SCS-Subtract:

for each object e_i in subtraction sequence
  if $e_{i_{front}} < z$
    $stencil \leftarrow 1$
  else
    $stencil \leftarrow 0$
  if $e_{i_{back}} > z$ and $stencil = 1$
    $z \leftarrow e_{i_{back}}$

(a) Draw initial 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 4: Convex Subtraction From Z-Buffer

Figure 4 parts (b) – (d) illustrate subtraction of a sphere from a rectangular block in the z-buffer. The OpenGL stencil buffer is used to flag pixels that are volumetrically inside the subtracted sphere. The z-buffer at these pixels is subsequently replaced with the back-facing surface of the subtracted sphere.

3.3 SCS-ZClip

Once subtraction is complete, pixels that have been completely subtracted are reset to Z_{far}. The z-buffer is compared to the back-facing surfaces of all of the intersected objects in the product. Pixels further than any back-facing surface of an intersected object are reset.

SCS-ZClip:

```
for each pixel
    stencil ← 0
for each intersected object, i
    if i_back < z
        stencil ← 1
for each pixel
    if stencil = 1
        z ← Z_far
    stencil ← 0
```

The z-buffer clipping step is illustrated in Figure 4 (e).

3.4 SCS Rendering Algorithm

The z-buffer of each CSG product is determined by applying the SCS-Intersect, SCS-Subtract and SCS-ZClip algorithms. If there are multiple products, a second z-buffer is used for storing the merged partial result. In a final pass, the shaded result is drawn into the colour buffer. The front-facing surfaces of intersected objects and the back-facing surfaces of subtracted objects are drawn with a z-equal depth test.

SCS:

```
for each pixel
    colour ← background
    z_merged ← Z_far

for each product
    for each pixel
        z_product ← Z_near
    SCS-Intersect
    SCS-Sequence
    SCS-Subtract
    SCS-ZClip
    for each pixel
        if z_product < z_merged
            z_merged ← z_product

for each intersected object, i
    if i_front = z_merged
        draw i_front
for each subtracted object, i
    if i_back = z_merged
        draw i_back
```

3.5 Discussion

The advantage of the SCS algorithm, compared to the Goldfeather [Goldf89, Stewa98] or Trickle [Epste89] algorithms is that a CSG product can be computed using a single z-buffer. Utilising multiple z-buffers by means of copying can be a significant bottleneck, [Goldf89, Wiega96, Stewa98, Stewa00] depending on the graphics hardware and available bandwidth. The SCS algorithm also makes use of linear-time processing of intersected convex objects, a strategy not found in other CSG rendering algorithms.

The disadvantage of a convex representation is that additional surface information may need to be rasterised in comparison to algorithms that can handle concave objects. The SCS algorithm is therefore most advantageous in the context of relatively high trian-

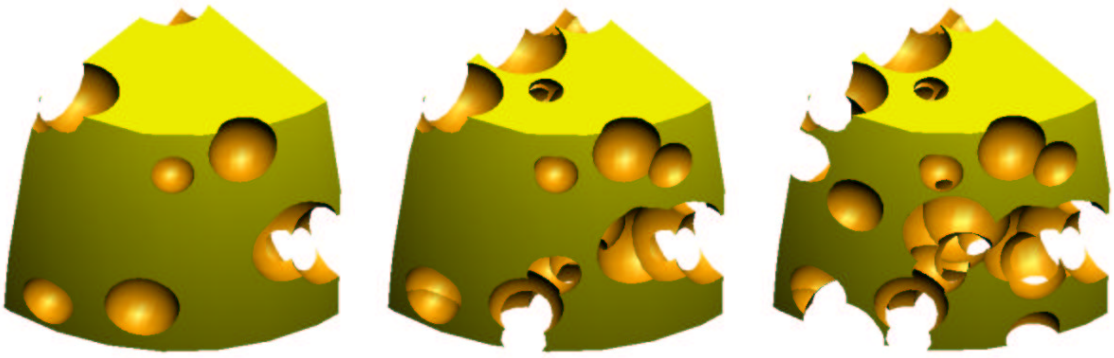
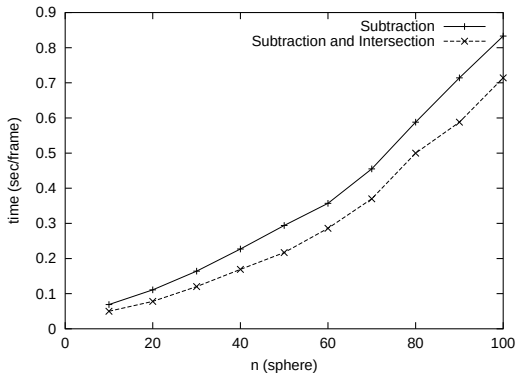


Figure 5: Swiss Cheese Model



n	Subtraction	Subtraction & Intersection	
	Frame Rate (frame/sec)	Frame Rate (frame/sec)	Speedup (%)
10	14	22	29
20	9.0	13	27
30	6.1	8.3	25
40	4.4	5.9	26
50	3.4	4.6	24
60	2.8	3.5	20
70	2.2	2.7	20
80	1.7	2.0	15
90	1.4	1.7	17
100	1.2	1.4	15

Table 2: SCS swiss-cheese performance.

gle rasterisation performance compared to z-buffer copying[Stewa00].

4 SCS TIMING RESULTS

A CSG test model was developed for verifying the performance advantage of the SCS-Intersect algorithm. The Swiss Cheese model in Figure 5 is formed by subtracting four boxes from an ellipsoid and randomly subtracting spherical holes of varied radius. This results in a CSG product consisting of $n+4$ subtractions: $ellipsoid - box_1 - box_2 - box_3 - box_4 - hole_1 - hole_2 - ... - hole_n$.

The same shape can be formed by intersecting boxes with the ellipsoid, rather than subtracting them. This alternative CSG tree consists of two intersections and n subtractions: $ellipsoid \cap box_4 \cap box_5 - hole_1 - hole_2 - ... - hole_n$

The average frame rates for the two CSG trees are given in Table 2. Time per frame is plotted in the corresponding graph.

The non-linear time requirement for CSG-Subtract is evident in the increasing slope of the graphs. The relative performance of CSG-Intersect and CSG-Subtract can be contrasted by comparing SCS-Intersect in Table 1 and SCS-Subtract in Table 2. For $n = 100$, there is an order of magnitude slowdown between 16 frame/sec for intersection and 1.4 frame/sec for subtraction.

The relative efficency of SCS-Intersect is also evident by contrasting the the performance of these two Swiss Cheese CSG models. The tree using intersection operations is between 15% and 30% faster to display. Some of this speedup is due to the $O(n)$ time intersection of boxes, rather than $O(n^2)$ time subtraction. Some of this speedup is also due to the fact that only two boxes need to be intersected, rather than four boxes being subtracted.

These results confirm that making use of the SCS-Intersect algorithm for intersections results in a performance improvement over a purely subtractive algorithm. They also suggest that intersection should be used (or even substituted, if possible) in preference to subtraction wherever possible.

5 CONCLUSION

The special case of intersection between finite convex objects is processed by the new linear-time SCS-Intersect rendering algorithm. The algorithm requires (roughly) three passes per object, and is therefore highly suitable for real-time interactive applications. SCS-Intersect requires one stencil buffer, one z-buffer and no z-buffer copying.

SCS-Intersect has been incorporated into the general pupose SCS (*Sequenced Convex Subtraction*) CSG rendering algorithm. Linear-time handling of intersections, rather than $O(n^2)$ time subtraction results in substantial speedup, depending on the relative number of intersections.

CSG products of around 100 convex objects can be displayed at interactive frame-rates at high resolution. NVIDIA GeForce3 hardware can display 100 intersected cylinders at approximately 7 frame/sec, and 50 subtracted spheres at approximately 5 frame/sec at 1024x768 resolution.

5.1 Future Work

We believe that there remains scope to further improve the performance of convex subtraction by producing shorter subtraction sequences. This can be facilitated by the analysis of adjacency (intersection) information between subtracted objects, without performing full depth-sorting for every frame.

5.2 Acknowledgements

This work was supported in part by the Co-Operative Research Center for Intelligent Manufacturing Systems & Technologies. This research arose from the C-5 collaborative research project involving ANCA Pty. Ltd., RMIT University and the CRC for IMST.

REFERENCES

[Boarda] OpenGL Arch. Review Board. *OpenGL Programming Guide*. Addison Wesley.

[Boardb] OpenGL Arch. Review Board. *OpenGL Reference Manual*. Addison Wesley.

[Epste89] D. Epstein, F. Jansen, and J. Rossignac. Z-buffer rendering from CSG: The trickle algorithm. *IBM Research Report RC 15182*, Nov 1989.

[Galbi76] G. Galbiati and F. P. Preparata. On permutation-embedding sequences. *SIAM J. of Appl. Math.*, 30(3):421–423, May 1976.

[Goldf86] J. Goldfeather, J. Hultquist, and H. Fuchs. Fast constructive solid geometry in the pixel-powers graphics system. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):107–116, Aug 1986.

[Goldf89] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. Near real-time CSG rendering using tree normalization and geometric pruning. *IEEE CG&A*, 9(3):20–28, May 1989.

[Kilga96] M. J. Kilgard. *The OpenGL Utility Toolkit (GLUT) Programming Interface*, 1996.

[Molna88] S. Molnar. Combining z-buffer engines for higher-speed rendering. *Proc. Eurographics '88, Third Workshop on Graphics Hardware*, pages 171–182, Sep 1988.

[Rappo97] A. Rappoport and S. Spitz. Interactive boolean operations for conceptual design of 3-d solids. *Computer Graphics (SIGGRAPH '97 Proceedings)*, 31:269–278, Aug 1997.

[Requi85] A. Requicha and H. Voelcker. Boolean operations in solid modelling: Boundary evaluation and merging algorithms. *Proc. of the IEEE*, 73(1):30–44, Jan 1985.

[Rossi94] J. R. Rossignac. Processing disjunctive forms directly from CSG graphs. *CSG 94: Set-theoretic Solid Modelling: Techniques and Applications*, pages 55–70, Apr 1994.

[Stewa98] N. Stewart, G. Leach, and S. John. An improved z-buffer CSG rendering algorithm. *1998 Eurographics/Siggraph Workshop on Graphics Hardware*, pages 25–30, Aug 1998.

[Stewa00] N. Stewart, G. Leach, and S. John. A z-buffer CSG rendering algorithm for convex objects. *The 8-th International Conference in Central Europe on Computer Graphics, Visualisation and Interactive Digital Media '2000 - WSCG 2000*, II:369–372, Feb 2000.

[Theoh01] T. Theoharis, G. Papaioannou, and E. Karabassi. The magic of the z-buffer: A survey. *The 9-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision - WSCG 2001*, Feb 2001.

[Wiega96] T. F. Wiegand. Interactive rendering of CSG models. *Computer Graphics Forum*, 15(4):249–261, Oct 1996.