Real-Time Animation of Objects Modelled using Constructive Solid Geometry

O'Loughlin, J
Dublin Institute of Technology

O'Sullivan, C.
Image Synthesis Group, Trinity College Dublin

ABSTRACT

Computer graphics have long been used to help people visualise complex entities such as machine components and assemblies. Systems such as Computer-Aided Design (CAD) applications allow the creation of models and designs for objects by providing interactive tools that can be manipulated by designers. An important facet of interactive design is that it should be possible to see the effects of changes made on models immediately, i.e. in real-time. In this paper we are interested in applications where renderings of complex objects, modelled using Constructive Solid Geometry (CSG), can be updated quickly enough to allow interactive use. The algorithms developed in this paper to allow real-time rendering of CSG-based objects require no specialised hardware or software to perform their tasks. They merely need a display adapter with stencil buffer support, and a graphics language capable of manipulating the colour, depth and stencil buffers provided by the display hardware. They can be implemented on low-cost PC-level systems and still prove capable of generating the required number of frames per second to allow real-time screen updates of CSG-modelled objects. Through empirical studies and observation we have identified that the algorithms developed here provide better performance than equivalent algorithms.

1 INTRODUCTION

This paper describes new algorithms for rendering solid objects modelled using Constructive Solid Geometry (CSG)[1]. CSG is a solid modelling technique used in applications such as Computer-Aided Design (CAD) to allow users to create complex solid objects from existing libraries of simpler objects. CSG uses Boolean operators to manipulate primitive solids (e.g. cubes and spheres) and assemblies of primitive solids. The Boolean operators used by CSG are:

- Union
- Intersection
- Subtraction

The data comprising a CSG model consists of a tree-like data structure containing nodes that describe the primitives to use and the Boolean operations to perform upon them (see Figure 1). The internal nodes of a "CSG tree" data structure store the CSG operations to be performed on the primitives stored at the leaf nodes.

[1] Refer to [Foley et al 90], [Hearn and Baker 94], [Vivo et al 92], [Su et at 91] and [Wyvill and Tosiyasu 85] for more information on CSG and its application.

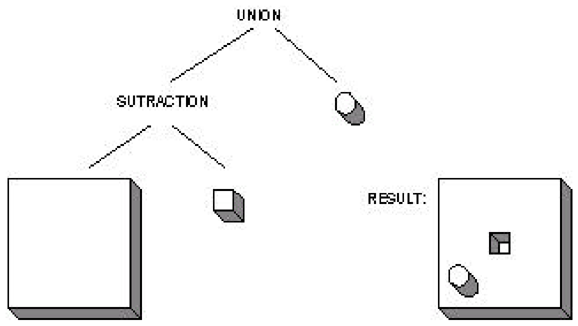


Figure 1: A typical CSG tree containing CSG operations in the internal nodes, and primitives in the leaf nodes.

The algorithms proposed in this paper render the effects of object intersection or subtraction on *two* primitive objects in a process we will call "Binary Composition"[2]. When rendering object intersection only the volume containing both objects simultaneously should be drawn, and for object subtraction the volume in one object that does not also contain the subtracted objects volume should be drawn. These algorithms must be capable of generating a certain number of frames per second to produce animated displays. The minimum number of frames per second generally accepted for smooth animations is ten, as with less than 10 FPS animations appear jerky and unpleasant to users. [Boyse and Gilchrist 82] propose rendering a CSG-based solid by first constructing another model of it using algorithms such as that from [Rossignac and Voelcker 89]. This creates a Boundary Representation, or B-Rep, model from the CSG model. The B-Rep is then rendered[3]. This is too slow for real-time animation on low-end hardware such as PC's, however, and would require dedicated graphics systems such as that proposed in [Nassar 92] for adequate performance. To reduce the processing overhead, and therefore take the best advantage of the available computing resources, several CSG rendering techniques propose rendering direct from the CSG tree, without generation of a secondary model. [Goldfeather et al 89], [Wiegand 96] and [McReynolds 96] all propose algorithms for object rendering directly from a CSG tree.

[2] The compositon of all the objects stored in a CSG tree we call "Complete Composition". Refer to [Wiegand 96] and [McReynolds 96] for details of how this can be achieved.
[3] Refer to [Pueyo and Mendoza 87] for another CSG rendering algorithm that uses a B-Rep secondary model.

We wish to create algorithms that are capable of rendering the effects of the intersection of two objects and the subtraction of one object from another. The algorithms must be capable of rendering enough frames per second to produce smooth real-time animations. There are two algorithms that need to be developed, one to compute and render the intersection of two objects, and another to compute and render the subtraction of one object from another. These problems are similar, but not the same. Therefore two separate algorithms have been developed, and each will be discussed. The algorithm for rendering object intersections will be called qAND (for quick AND), and the other algorithm is called qSUB (for quick SUB). An AND operation is an object intersection, and a SUB operation is a subtraction of one object from another. This paper describes these algorithms, and examines their performance relative to existing algorithms. Tests were performed to examine the performance of the proposed qAND and qSUB algorithms, and on existing AND and SUB algorithms to allow for a comparison between the existing and the proposed algorithms.

2 THE NEW ALGORITHMS

There are a number of possible approaches to rendering CSG-based objects, including the use of basic geometric object data, ray tracing and use of stencil buffers. The algorithms described here use the latter approach because it provides the best performance, and this is important for applications requiring object animation. The qAND and qSUB algorithms use three buffers supported in display hardware. They are:

- ***The Colour Buffer:*** A colour buffer stores the colour information for each pixel in a frame. There can be more than one of these, for example there can be separate colour buffer for red, green and blue components of an image.
- ***The Depth Buffer:*** A depth buffer stores the depth information for each point on the surface of an object in a frame. This information consists of the location on the Z-axis of each point.
- ***The Stencil Buffer:*** A stencil buffer acts as a mask, determining what pixels in a frame can be written to, and which cannot. Objects are drawn into a stencil buffer and then other objects are clipped against the information in the stencil buffer to determine what parts of them should actually be displayed on-screen, i.e. written into the colour buffers.

2.1 The qAND Algorithm

The qAND operation renders on-screen only the volume containing both objects in a binary composition simultaneously, i.e. A•B, or the object intersection. If we consider the process of rendering the qAND of two objects called, say, A and B, this problem consists of two steps:

- Find the parts of A's volume that exist inside B's volume, and draw them on-screen, and
- Find the parts of B's volume that exist inside A's volume, and draw them on-screen.

The steps above, therefore, require that the qAND algorithm be capable of finding any parts of the surface of A that exist within the volume of B and drawing those parts on-screen, and of finding the parts of the surface of B that exist within the volume of A and drawing them also. The combination of these two sets of surface points is the qAND of the two objects.

The algorithm for qAND involves writing the front faces of A into the stencil and depth buffers, but not the colour buffers. Then the front faces of the other object, B, are written into the colour buffers, but only where points on the front faces of B fail the depth test against the front faces of A *and* where A has been written into the stencil buffer. This has the effect of rendering any any surfaces of B that are inside A. The procedure is then repeated, but in this second pass the objects are reversed so that the surfaces of A that are inside the volume of B are rendered into the colour buffers. The composition of the writes from the two passes constitutes the rendering of the intersection of the two objects.

Figure 2 illustrates the different rendering phases in the qAND algorithm. Steps 1 and 2 discover what parts of object B lie inside the volume of object A, and render the surfaces of B inside this volume into the colour buffer. Steps 3 and 4 perform exactly the same operation, except this time objects A and B are swapped so that the parts of A inside B are discovered and rendered. The final image is created by this composition of two sets of surfaces into the colour buffer. The first set is written into the colour buffer at step 2 and the second set is written at step 4. In total only 4 writes to display buffers are required to produce a rendering of the intersection of two objects (the writes to the depth and stencil buffers in steps 1 and 3 are performed simultaneously).

2.2 The qSUB Algorithm

The qSUB operation is slightly more complex than qAND. The fundamental requirement of qSUB is to render only the parts of one object that do not lie inside the volume of another object, i.e. to render only the intersection of one object with the complement of another object. However this approach would be too simplistic, as it also proves necessary to find any parts of the subtracted object that should appear because they lie inside the volume of the other object and should be visible to the user to create the desired effect. The two phases required in qSUB are:

1. Find the parts of the subtracted object that exist inside the object being subtracted from and render them.
2. Find those parts of the object being subtract from that should be visible to the user, and render them.

If we are subtracting an object B from an object A then the first phase in the qSUB algorithm involves drawing the back faces of A into the depth and stencil buffers only, i.e. not into the colour buffers. The back faces of B are then drawn into the colour buffers where they pass the depth test against A *and* A exists in the stencil buffer.At this stage any back faces of B that exist inside A's volume will have been rendered into the colour buffers. The next step is to draw the front-facing surfaces of A into the colour buffer also to complete the image. To achieve this this front faces of B are drawn into the depth and stencil buffers only, and then the front faces of A are drawn into the colour buffers where they pass the depth test against B. This second phase to the algorithm has drawn in the front faces of A where back facing surfaces of B are not present, completing the rendering of the overall qSUBed object.
Figure 3 illustrates the phases in the qSUB algorithm. The parts of the object B are found, which are inside another object A, from which B is being subtracted. Only those parts of B that might need to be displayed to the user are actually rendered into the colour buffer. This occurs at step 2. Steps 3 and 4 find the parts of A that should be displayed to the user, and render them into the colour buffer also. This composites the two sets of writes and completes the image. Once again only four writes are required to display buffers to achieve the desired effect.

2.3 Additional Requirement for qAND and qSUB

The qAND and qSUB algorithms described in this chapter have an additional requirement for usage not present for the AND and SUB algorithms. If two objects that are non-intersecting in the Z-axis are combined using the AND and SUB algorithms then the correct result will be generated, i.e. nothing for AND, and the entire non-subtracted object for SUB. The qAND and qSUB algorithms require an extra step to correctly render the results for this scenario. The extra step is the use of a collision detection mechanism such as that provided in [O'Sullivan 96] to decide whether or not to render a result on-screen. If two objects do not intersect in the Z-axis then nothing should be drawn. The new algorithms work correctly for objects that intersect on the X or Y-axis without this added step.

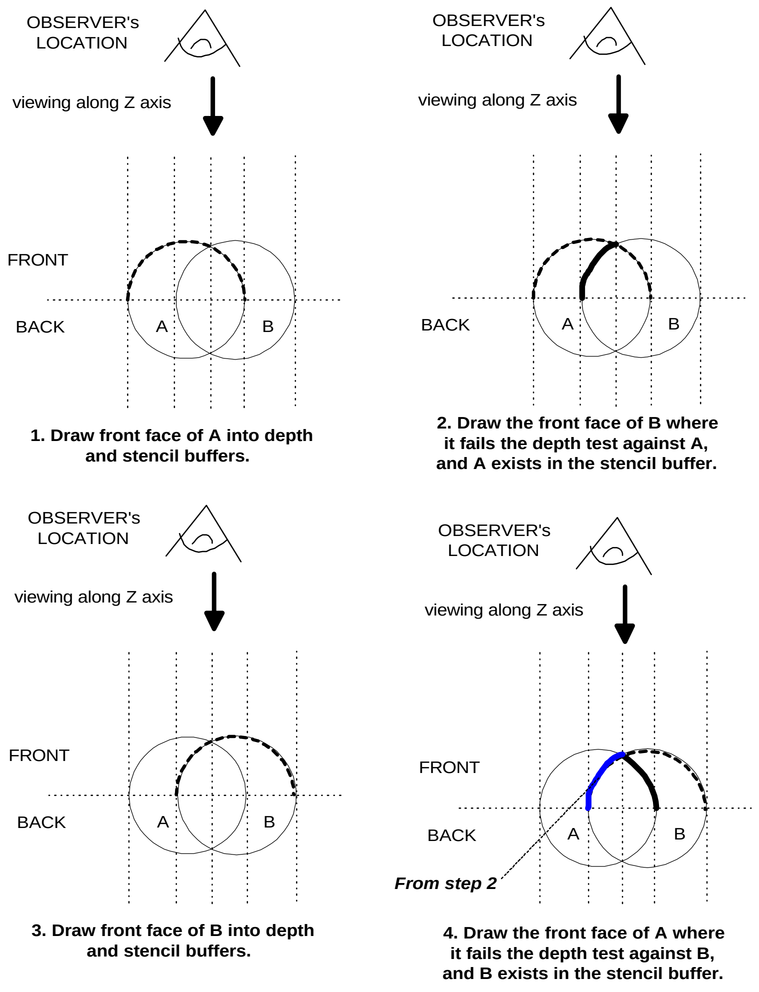

OBSERVER's LOCATION
viewing along Z axis
FRONT
BACK
A
B
1. Draw front face of A into depth and stencil buffers.
OBSERVER's LOCATION
viewing along Z axis
BACK
A
B
2. Draw the front face of B where it fails the depth test against A, and A exists in the stencil buffer.
OBSERVER's LOCATION
viewing along Z axis
FRONT
BACK
A
B
3. Draw front face of B into depth and stencil buffers.
OBSERVER's LOCATION
viewing along Z axis
FRONT
BACK
A
B
From step 2
4. Draw the front face of A where it fails the depth test against B, and B exists in the stencil buffer.

Figure 2: The Phases of the qAND algorithm

3 PERFORMANCE

The qAND and qSUB algorithms were derived from existing algorithms in [McReynolds 96]. The intersection and subtraction algorithms therein both require 9 writes to the three buffers used, i.e. colour, depth and stencil, to produce the required results. Therefore a series of tests was performed to determine the performance characteristics of both pairs of intersection and subtraction algorithms, and their relative performance. For each test the ability of the algorithms to produce real-time animations, i.e. at least 10 FPS, was considered. A total of 23 different tests were run. These were divided into two phases:

1. Determination of the performance of the underlying graphics system when rendering primitive objects on-screen.
2. Determination of the performance of the intersection and subtraction algorithms when rendering complex objects on-screen.

Each test was run several times with different primitive sizes and combinations being rendered in each run. The results were collected and compared to determine the effects of factors such as primitive complexity, size and motion on the ability of the algorithms to produce real-time results. Table 1 contains a list of the tests performed, the operations carried out in each test and the primitives used.

The operations performed are called ONE, OR, AND or qAND and SUB or qSUB, and the results are shown in Figure 4. ONE simply involves drawing a primitive into the colour buffer with depth testing enabled. OR draws two objects into the depth buffer, again with depth testing enabled. AND and SUB are the names we use for the intersection and subtraction algorithms from [McReynolds 96], and qAND and qSUB are the new intersection and subtraction algorithms proposed in this paper. Tests 8 to 23 involved rendering an animated sequence rendered using first the qAND or qSUB algorithm, and then the AND or SUB algorithm, respectively. This allowed a direct comparison to be performed between the two pairs of algorithms. The tests were run on a Dell Latitude CPi notebook PC, with a 266MHz Pentium II processor, 64Mb of RAM and a Neomagic 128XD display adapter with 2Mb of video RAM.

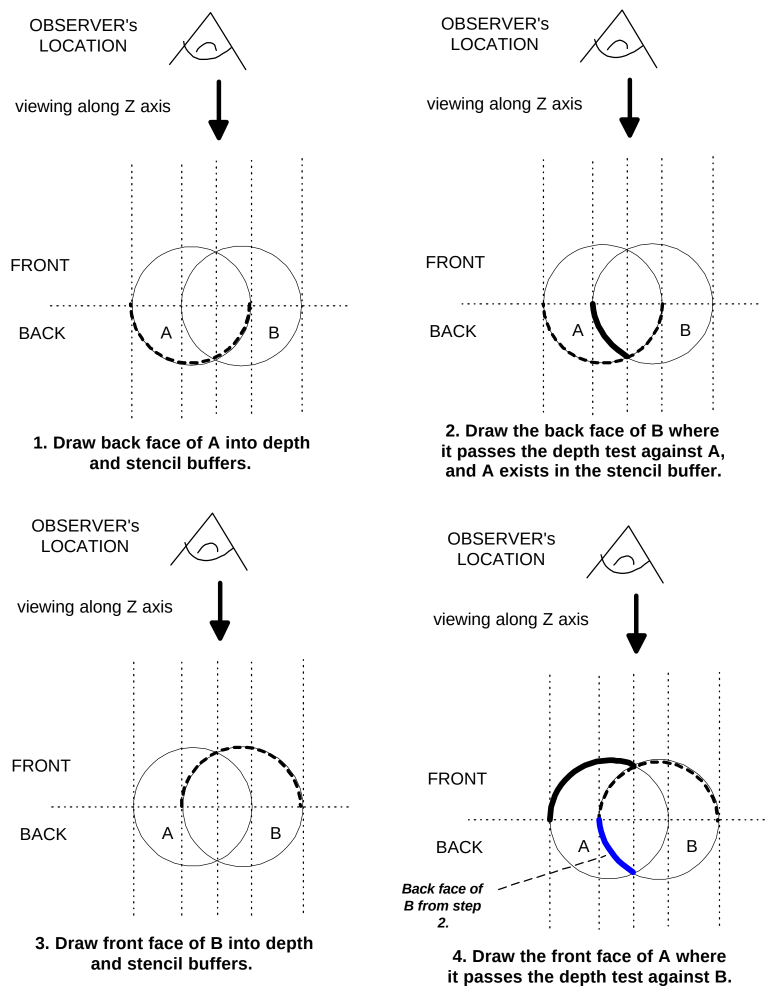

OBSERVER's LOCATION
viewing along Z axis
FRONT
BACK
A
B
1. Draw back face of A into depth and stencil buffers.
2. Draw the back face of B where it passes the depth test against A, and A exists in the stencil buffer.
3. Draw front face of B into depth and stencil buffers.
Back face of B from step 2.
4. Draw the front face of A where it passes the depth test against B.

Figure 3: The Phases of the qSUB algorithm.

Test	Operations	Description
1	ONE	Cube with size 1 unit (cube size is relative to window size) drawn into variable size window.
2	ONE	Fixed size cube drawn into variable size window. Cube size remains the same for each window size.
3	ONE	Variable-sized cube drawn into fixed-size window.
4	ONE	Variable-sized sphere drawn into fixed-size window.
5	OR	Two variable-size cubes drawn into fixed-size window.
6	OR	Two variable-size spheres drawn into fixed-size window.
7	OR	Variable-size cube and sphere drawn into fixed-size window.
8,9	qAND, AND	Cube-Cube Intersection.
10,11	qAND, AND	Sphere-Sphere Intersection.
12,13	qAND, AND	Cube-Sphere Intersection.
14,15	qSUB, SUB	Cube-Cube Subtraction.
16, 17	qSUB, SUB	Sphere-Sphere Subtraction.
18, 19	qSUB, SUB	Cube-Sphere Subtraction.
20, 21	qAND, AND	Cube-Cube Intersection, with cube rotation.
22, 23	qSUB, SUB	Cube-Cube Subtraction, with cube rotation.

Table 1: Tests Performed

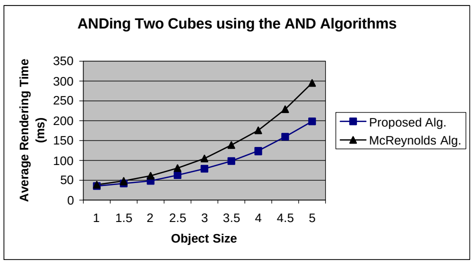
The first set of tests, from 1 to 7, proved that the underlying graphic subsystem consisting of the combination of the OpenGL graphics library used to render the tests and the display hardware were fast enough at rendering primitive solid objects such as spheres and cubes on-screen to support the real-time rendering of more complex objects, i.e. those produced via qAND and qSUB.

In the remaining tests a direct comparison between the existing AND and SUB algorithms, and the qAND and qSUB algorithms presented in this paper showed that the new algorithms were more efficient than the older ones, and could thus render more frames per second. The results of these tests are shown in Table 1.

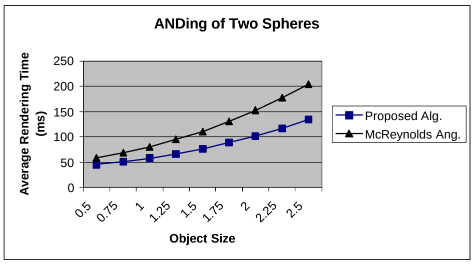
4 CONCLUSIONS AND FUTURE WORK

The tests performed upon the proposed and existing CSG algorithms prove a number of points. The proposed qAND algorithm offers better rendering performance than the AND algorithm. It consistently renders more frames per second than the older algorithm does, and so it is more appropriate for real-time animation applications. The proposed qSUB algorithm also shows better performance than the SUB algorithm. For larger objects it proves to be up to twice as fast as the existing SUB algorithm. The proposed qAND and qSUB algorithms show little performance degradation when asked to render objects with translational and/or rotational velocities. Since each frame in an animation is generated from scratch, with no previous frame information being reused, changes in object location or rotation are handled as a normal occurrence by the proposed algorithms. The little overhead that does appear in, for example, rendering rotating objects can be attributed to the amount of time taken to apply the rotation matrix to the object’s description. A general conclusion that can be made from the testing of the new algorithms is that they do provide better overall rendering performance than the older algorithms, and that, allowing for certain limitations such as object size, they are capable of generating real-time animations of CSG-based objects.

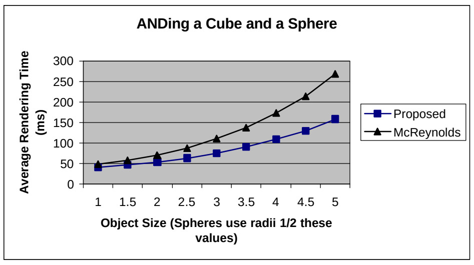
Much scope for future research remains however, the qAND and qSUB algorithms developed in this dissertation leave side-effects such as unwanted information about object surface depths in depth buffers. If a complete object composition is to be performed from a CSG tree then each binary composition performed at the leaves of the tree must be composited together according to the CSG operations at the internal nodes of the tree to produce a rendering of the complete object. Thus the next problem to be solved, after the binary composition problem, is that of the complete composition problem. The side-effects left behind by each binary composition must be overcome so that each binary composition can be rendered correctly into the available colour and depth buffers. The stencil buffer information can generally be ignored between binary compositions as it only has an impact on the pair of shapes that are being composited together. The new algorithms for qAND and qSUB proposed in this dissertation could be extended to encompass the rendering of entire CSG trees modelling complex objects. To allow this no unnecessary information could be left in depth or stencil buffers after binary composition of objects. This would also require a multi-pass approach.



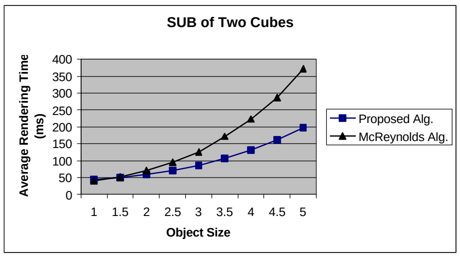
Results of Tests 8 and 9



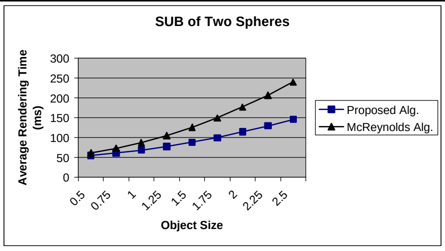
Results of Tests 10 and 11



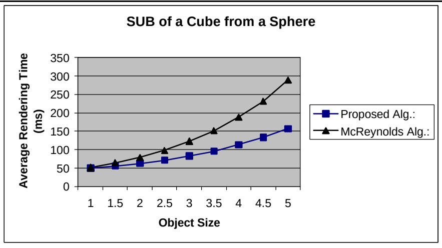
Results of Tests 12 and 13



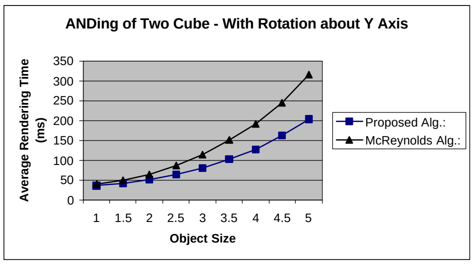
Results of Tests 14 and 15



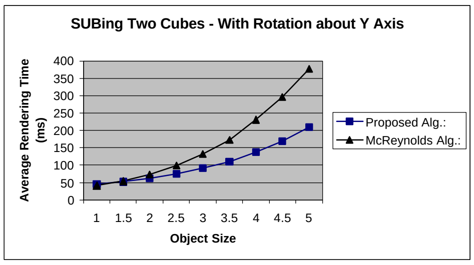
Results of Tests 16 and 17



Results of Tests 18 and 19



Results of Tests 20 and 21



Results of Tests 22 and 23

Table 2: Test Results

References

[Boyse and Glichrist 82] “GMSolid: Interactive modelling for design and analysis of Solids”, J.W. Boyse & J.E. Gilchrist, CG & A, 2(2), March 1982, P.27-40.

[Foley et al 90] “Computer Graphics: Principles and Practice”, Foley & van Dam & Feiner & Hughes, Addison Wesley, 1990. ISBN: 0-201-12110-7.

[Goldfeather et al 89] “Near Real-Time CSG Rendering using Tree Normalisation and Geometric Pruning”, Jack Goldfeather & Steven Molnar & Greg Turk & Henry Fuchs, IEEE Computer Graphics and Applications, Vol. 9 Pt. 3, P.20-28, 1989.

[Hearn and Baker 94] “Computer Graphics”, Hearn Donald & Baker M. Pauline, Prentice Hall International Editions, ISBN: 0-13-159690-X, 1994.

[**McReynolds 96]** McReynolds, T. "Programming with OpenGL: Advanced Rendering" SIGGRAPH ’96 Course, New Orleans, USA.
URL: http://www.sgi.com/software/opengl/advanced96/

[Nassar 92] “Fast Display of Solid Objects”, Nassar M., Journal Of Microcomputer Applications, 1992, Vol. 15 July, P. 187-193.

[O’Sullivan 96] “Real-time Collision Detection for Computer Graphics”, O’Sullivan Carol, Dissertation, Dublin City University, August 1996.

[Pueyo and Mendoza 87] "A New Scan Line Algorithm for the Rendering of CSG Trees", Xavier Pueyo & Joan Carles Mendoza, Eurographics '87, 1987.

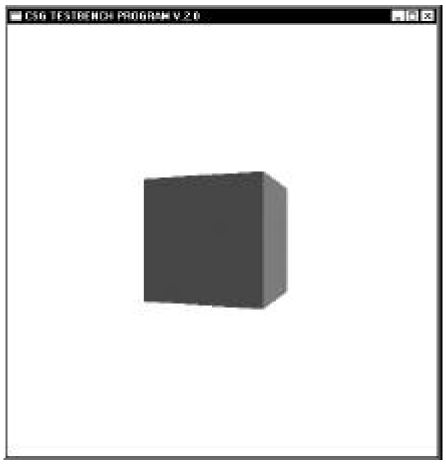
[Rossignac and Voelcker 89] "Active Zones for in CSG for Accelerating Boundary Evaluation, Redundancy Elimination, Interference Detection and Shading Algorithms", J.R. Rossignac & H. Voelcker, ACM TOG, 8(1), January 1989, P.51-87.

[Su et al 91] "Generalised CSG: A Solid Modelling Basis for High Productivity CAD Systems", Dr. Chuan Jun Su & Dr. Richard J. Mayer & David C. Browne, Autofact 91 Conference Proceedings, 1991.

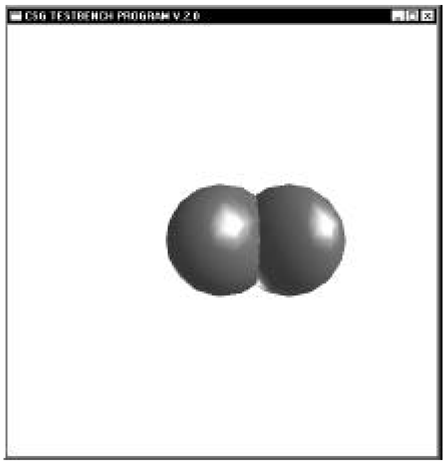
[Wiegand 96] "Interactive Rendering of CSG Models", Wiegand T.F., Computer Graphics Forum, 1996, Volume 15 No. 4, PP. 249-261.

[Wyvill and Tosiyasu 85] "A functional model for Constructive Solid Geometry", Geoff Wywill & Tosiyasu L. Kunii, The Visual Computer, Vol.1 Pt.1, July 1985, P.3-14.

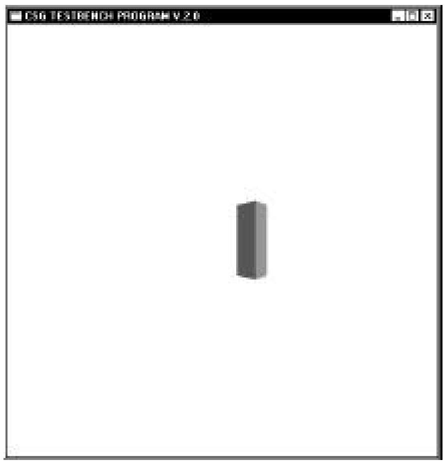
[Vivo et al 92] "A Secondary Parametric Model for CSG", Roberto Vivo & Emilio Camahort & Ricardo Quiros, Computing and Graphics, Vol. 16 No.4, P. 369-373, 1992.



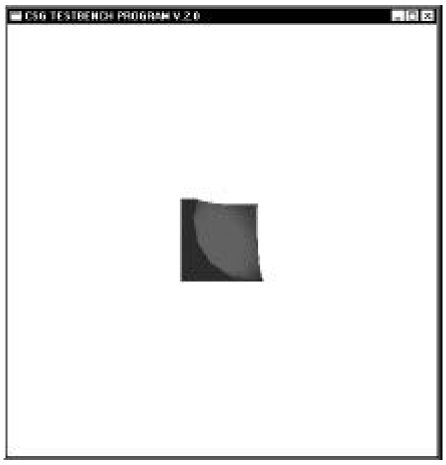
ONE operation draws a primitive object (a cube).



OR operation draws the union of two spheres.



qAND operation draws the intersection of two cubes. The cubes were both rotated by 20 degrees about the Y-axis to produce the "wedge" shape shown.



qSUB operation subtracts a sphere from a cube and draws the result. The sphere was located up and to the right of the cubes centre point, and slightly closer to the observer's position.

Figure 4: Application Screens