AN IMAGE-SPACE ALGORITHM FOR HARDWARE-BASED RENDERING OF CONSTRUCTIVE SOLID GEOMETRY

Nigel Timothy Stewart

Bachelor of Applied Science in Computer Science

Submitted in fulfillment of the requirements of
the degree of Doctor of Philosophy

May 2008

School of Aerospace, Mechanical and Manufacturing Engineering
Science, Engineering and Technology Portfolio
RMIT University

CANDIDATE DECLARATION

This work is that of the author, except where due acknowledgement has been made. This work has not been submitted previously, in whole or in part, to qualify for any other academic award. The content of this thesis is the result of work carried out in the course of an approved research program since the official commencement date. Paid and unpaid editorial work carried out by third parties has been acknowledged.

______________________________ ____________________

Nigel T. Stewart Date

Table of Contents

List of Figures

List of Tables

List of Algorithms

Glossary

$-$	Set Difference notation
$\cap$	Set Intersection notation
$\cup$	Set Union notation
Acyclic Graph	A graph containing no cycles
A_n	The unordered set of n elements $\{a, b, \cdots, n\}$
API	Application Programming Interface
CNC	Computer Numerically Controlled (machinery)
CPU	Central Processing Unit
CSG	Constructive Solid Geometry
CSG Tree	Primitive shapes and CSG operators such as union, intersection and difference arranged in a tree
Cyclic Graph	A graph containing one or more cycles
Depth Buffer	Per-pixel storage of depth in the frame buffer
Depth Test	Testing of rasterised fragment depth with respect to depth buffer
Difference	Also known as Subtraction
Disconnected Graph	A graph consisting of separate connected graphs
Empty Intersection	The intersection of volumes that do not overlap is empty.
Fragment	Pixel sized geometry fragment produced by rasterisation, denoted F
Frame Buffer	Graphics hardware per-pixel memory including colour, stencil and depth
Goldfeather	CSG rendering algorithm [36, 37]
GPU	Graphics Processing Unit
Intersection	Volumetric boolean intersection of volumes, denoted $\cap$
k-permutation	Sequence of k unique elements selected from n
kPES	k-Permutation Embedding Sequence
k-tuple	Sequence of k elements
Leaf Node	A graph node connected to only one other
Leaf Trimming	The process of removing leaf nodes from a graph

L_m	A permutation embedding sequence of length m
$\mathcal{L}(n)$	The set of permutation embedding sequences of A_n
NNR	Normalised No-Repeat sequence
NNR-size	The number of unique NNR sequences of a particular length
Normalised PES	A permutation embedding sequence in normalised form
OpenGL	2D and 3D real-time graphics API (Application Programming Interface)
Overlap Graph	An undirected graph of spatial overlap between shapes
Permutation	Sequence of n unique elements
PES	Permutation Embedding Sequence
Primitive	3D shape used in CSG modelling
Rasterisation	Conversion of geometry to fragments
Ring Graph	A cyclic graph consisting of a ring
SCS	CSG rendering algorithm [88, 89, 90]
Sequence Encoding	The process of forming a permutation embedding sequence (PES)
Shortest Length PES	A permutation embedding sequence of minimal length
$\mathcal{S}_n$	The the set of permutations of A_n
Stencil Buffer	Integer per-pixel storage in the frame buffer.
Stencil Test	Testing of a rasterised fragment with respect to stencil buffer.
Subtraction	Volumetric boolean subtraction of volumes, denoted $-$
Trickle	CSG rendering algorithm [28]
Union	Volumetric boolean union of volumes, denoted $\cup$
Z-Buffer	Also known as Depth Buffer
z-less	In graphics hardware depth-testing, test that incoming fragments are closer (less than) the depth stored in the z-buffer
z_{far}	Far clip plane depth value
z_{near}	Near clip plane depth value
z_F	Fragment depth value

Summary

This work investigates computer graphics techniques for image-space hardware-based rendering of objects composed of boolean combinations of three dimensional objects. In the field of computer graphics this is called *Constructive Solid Geometry* (CSG) Rendering.

This work is motivated by the evolving functionality and performance of computer graphics hardware. These platforms are optimised for interactive computer graphics rendering rather than being intended as general purpose processors such as CPUs. The dramatic increase in the computational capacity of graphics hardware creates the opportunity for graphics hardware image-space approaches to problems previously solved using CPU object-space approaches. One attraction of the image-space approach is that the stream processing architecture of graphics hardware facilitates parallelisation in a transparent and scalable manner.

The functionality and design of graphics hardware supports a variety of approaches to CSG rendering as reported in the literature. New techniques have also been developed in the course of this work. The overall goal is to minimise the amount of time required to render each frame in an animation or interactive application involving boolean combinations of shapes.

This work is also motivated by a specific industrial application — interactive verification of five axis grinding machine tool programs. The movement of a workpiece and abrasive wheel result in a complex cutting tool that is intended to achieve industrial requirements such as shape, sharpness, strength, durability and aesthetics.

CSG rendering techniques provide a means of previewing the final shape of a tool with computer graphics as an alternative to using the actual manufacturing process. Visual simulation can provide functionality that would be difficult in the real world. The movement of the abrasive wheel can be animated and interactively inspected without the obstruction of liquid coolant that is needed to dissipate heat. The speed of an animation can be adjusted or paused independent of mechanical limitations. Distances and angles which can be difficult to measure, particularly during manufacturing, can be measured. Simulation is also substantially cheaper than operation of the machinery.

The CSG rendering algorithm developed in this work takes the approach of optimising performance for combinations of convex shapes. Concave shapes must be decomposed into convex shapes for the purpose of rendering. This results in an algorithm better suited to modern graphics hardware with improved interactivity. Our rationale is analogous to polygon rasterisation — graphics hardware tends to be limited to convex planar polygons or triangles, relying on CPU-based conversion from concave or non-planar polygons.

This new approach to CSG rendering also raises new questions, issues and challenges related to CSG rendering. We introduce the concept of CSG subtraction sequences, related to the mathematical concept of a *Permutation Embedding Sequence* (PES). We show these sequences to be $O(n^2)$ in length.

Our work also introduces the concept of overlap graph based subtraction sequence encoding. This approach results in $O(n)$ to $O(n^2)$ sequences, depending on the spatial arrangement of subtracted objects.

Our new algorithm called *Sequenced Convex Subtraction* (SCS) has been implemented and verified on current computer systems. Our work has also examined the performance of SCS relative to previous algorithms, as well of performance aspects of the SCS algorithm itself.

Acknowledgements

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

Early work towards the SCS CSG rendering algorithm was in the commercial setting of ANCA Pty Ltd, an Australian manufacturer of five axis tool and cutter grinding machines. I am grateful to Glenn Brien and Mike Simakov for the opportunity to be part of the Cimulator3D project team, and ANCA for pioneering the NC verification technology niche for five axis machining. I would also like to thank my co-workers at ANCA for their friendship and professionalism.

Collaboration with Robert Erra of ESIEA Paris, and Nik Lygeros of Université Claude Bernard Lyon 1 resulted from our mutual interest in permutation embedding sequences. I was pleased to be able to apply this to CSG rendering having grappled with the mathematical concepts and formulation.

This work would not have been possible without the advice, encouragement, support and input of my academic supervisors at RMIT University: Dr Sabu John of the School of Aerospace, Mechanical and Manufacturing Engineering; Geoff Leach of the School of Computer Science and Information Technology.

I would like to thank my examiners for sparing the time and energy to review this work so closely and the thoughtfulness of their valuable feedback.

Over the course of my postgraduate studies my wife Fiona Smith has been a supporter, nutrition and health consultant and coach, proof-reader, travel and life companion, motivator, and always a reference point beyond the commercial and academic technical realm. Spanning three countries, many years, and the birth of our daughter Talia the course of my studies and career has been more interesting and wonderful than I could have imagined.

Many freely available software packages were used in the course of this research. These include D.E. Knuth's LaTeX (and associated tools) for typesetting, the GNU GCC C/C++ compiler and command line tools by the Free Software Foundation, Shawn Yarbrough's Purenum arbitrary precision integer library, Graphviz graph drawing software by AT&T Research, the GAlib genetic algorithm library by Massachusetts Institute of Technology, the GIMP image processing application, the XEmacs and Kate text editors, the POV-Ray raytracing rendering application, the Linux operating system, the KDE desktop environment, and many others.

I have found Wikipedia to be a remarkably complete and upto date secondary source of information, competing with google as my Firefox start page.

Austin, Texas, USA
Nigel Timothy Stewart May 21st, 2008

Chapter 1

Introduction

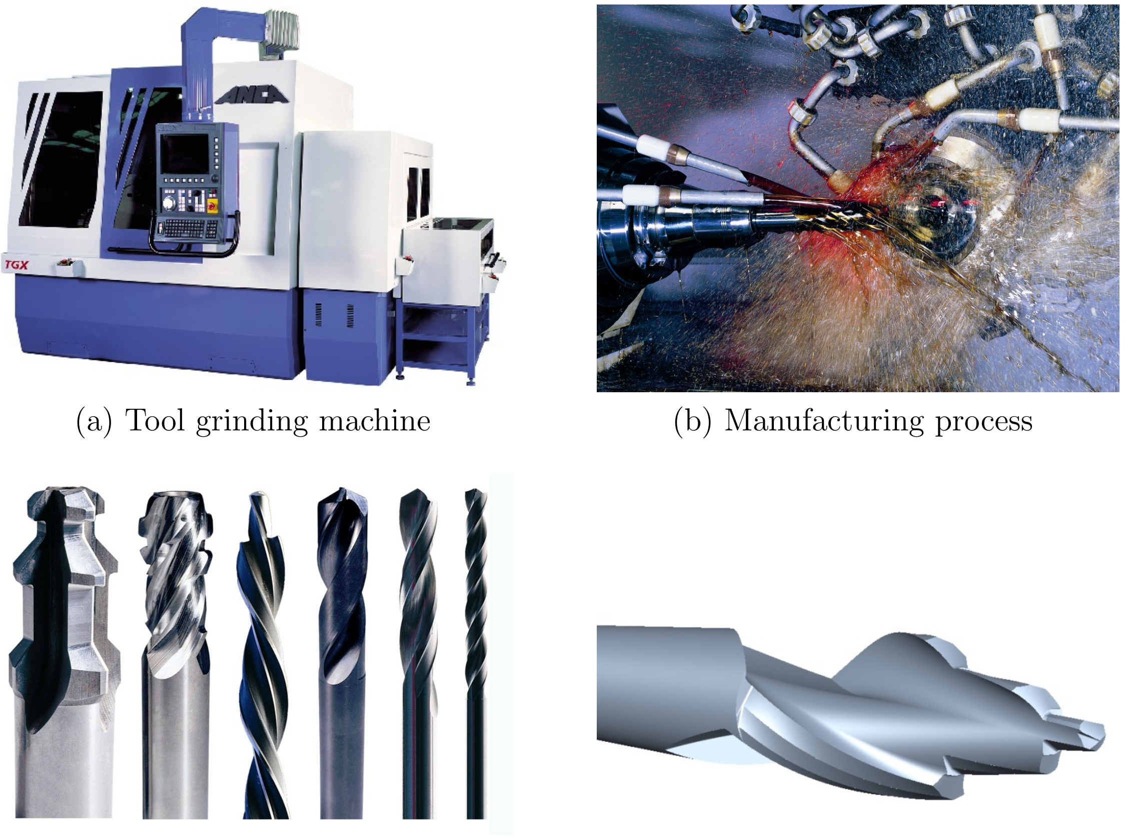
What we call 'Progress' is the exchange of one nuisance for another nuisance.

— Havelock Ellis

1.1 Motivation

Five axis *Computer Numerically Controlled* (CNC) grinding machines are used for the manufacture of drills, mills and other cutting tools, as illustrated in Figure 1.1. An abrasive grinding wheel with five degrees of freedom is moved through space resulting in material being removed from a solid workpiece. The final shape is determined by the volumes swept by the wheel over time. The motivation for this project is the simulation of the tool grinding manufacturing process to improve tool design and development.

Computer based simulation of tool manufacturing involves the use of algorithms for volumetric subtraction. Image-based approaches are considered appropriate for visualisation and design purposes due to their relative interactivity, flexibility and potential performance. Previous image-space algorithms [28, 37, 97] for *Constructive Solid Geometry* (CSG) were investigated, but found to be ill-suited to five-axis CNC tool path verification on graphics hardware available at that time[1].



(a) Tool grinding machine

(b) Manufacturing process

(c) Machine tools

(d) Computer generated image of a Step Tool

Figure 1.1: Numerically Controlled Manufacturing of Cutting Tools

[1] SGI Indy and PC 3Demon graphics hardware in 1996.

Since the introduction of the first commodity graphics accelerators there has been dramatic improvement in price, performance and functionality. This work is motivated in part by the prospect of utilising 3D hardware for image-space CSG rendering — a generalisation of the visible surface problem. We believe that the feasibility of real-time interactive CSG rendering depends on further improvements to performance, functionality and programmability at the hardware level, as well as improved algorithms. The focus of this work is CSG rendering algorithms suited to the OpenGL 1 fixed pipeline architecture and the corresponding generation of graphics hardware.

This research originated in the context of an industry-based research program involving ANCA Pty Ltd, an Australian tool grinding machine company. ANCA's five-axis tool grinding simulation package Cimulator3D [69] was initially released in late 1997 and was the first of its kind in the tool grinding industry. The work in this thesis using an image-space approach represents a divergence from the implementation of Cimulator3D which uses an object-space approach. Both approaches aim to solve the same technical challenge — fast and accurate subtraction of complicated swept volumes.

1.2 Contribution

In this work an alternative approach to image-space CSG rendering is formulated, based on sequenced subtraction of convex objects. A new algorithm is described for CSG rendering that we call *Sequenced Convex Subtraction* (SCS). The algorithm minimises graphics hardware buffer copying by focusing on subtraction of only convex objects. Subtraction is repeated for individual objects until the complete subtracted result is formed in the depth buffer. Other approaches to CSG rendering have relied on either sorting subtracted geometry or merging simpler subtracted results.

This work also examines the interesting and challenging problem of forming shortest length subtraction sequences. Several practical[2] subtraction sequence encoding methods are used in our implementation of the SCS CSG rendering algorithm. Shortest length subtraction sequences can also be utilised in some cases, but can not yet be determined in general.

The main contribution of this work is the Sequenced Convex Subtraction algorithm for CSG Rendering. The most advantageous aspect of this algorithm is the efficient subtraction of large numbers of relatively dispersed convex objects.

[2] The sequence encoding methods are practical in the sense that they are quickly and easily computed, but result in sequences that may be longer than absolutely necessary.

1.3 Publications

Peer-reviewed publications relating to this work:

- N. Stewart, G. Leach, S. John, *An Improved Z-Buffer CSG Rendering Algorithm*, 1998 Eurographics/Siggraph Workshop on Graphics Hardware, pp. 25-30
- N. Stewart, G. Leach, S. John, *A CSG Rendering Algorithm for Convex Objects*, The 8th International Conference in Central Europe on Computer Graphics, Visualisation and Interactive Digital Media 2000 — WSCG 2000, Volume II, pp. 369-372
- R. Erra, N. Lygeros, N. Stewart, *On Minimal Strings Containing the Elements of S_n by Decimation*, Discrete Mathematics & Theoretical Computer Science, Proceedings vol. AA (2001), pp. 165-176
- N. Stewart, G. Leach, S. John, *Linear-time CSG Rendering of Intersected Convex Objects*, The 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2002 — WSCG 2002, Volume II, pp. 437-444
- N. Stewart, G. Leach, S. John, *Improved CSG Rendering using Overlap Graph Subtraction Sequences*, Graphite 2003 — International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia, pp. 47-53

1.4 Methodology

The techniques developed in this work have followed an iterative process of gathering requirements, researching and implementing known approaches, conception and implementation of performance improvements, and publication of the results. Over time, a significant codebase of supporting functionality was designed, developed and adapted to new ideas and requirements.

The approach, publications and implementation discussed here diverged from the commercial context early in the project, although it was structured to be suitable for reintegration and deployment. In this manner ANCA's commercial codebase and intellectual property were quarantined from academic disclosure. This also provided flexibility in developing and deploying the algorithms on platforms and hardware outside of ANCA Pty Ltd.

Our work also involved testing performance and accuracy of the algorithm in a variety of situations. This information was in turn used as a basis for further development and improvement of the algorithm. The specific requirements for CNC verification of tool grinding were held as the guiding priority, although our efforts included abstracting and generalising the results as much as possible.

Core portions of the final code base appear in Appendix B, and also online [8].

1.5 Thesis Overview

Various aspects of the SCS CSG rendering algorithm have been divided into the following six chapters of this thesis. In this work algorithms are formulated in terms of OpenGL functionality and terminology.

Chapter 2 reviews the CSG approach to geometric modelling including tree traversal, transformation and pruning algorithms.

Chapter 3 introduces the pixel processing capabilities of the OpenGL *Application Programming Interface* (API) relevant to this work including z-buffer and stencil testing. Previous CSG rendering techniques are surveyed, including underlying concepts and previous approaches such as the Trickle [28] and Goldfeather [37, 97] algorithms.

Chapter 4 presents our new *Sequenced Convex Subtraction* (SCS) CSG rendering algorithm. The algorithm is based on the intersection and subtraction of convex objects. Subtracted objects are encoded into a subtraction sequence that ensures the correctly rendered result. Simple practical subtraction sequence encoding algorithms are described.

Chapter 5 abstractly formulates a subtraction sequence as a *Permutation Embedding Sequence* (PES). Related problems in mathematics and cryptography are discussed. A list of known shortest length PESs is presented. Techniques for efficiently searching for shortest length PESs are also described. A proof of $O(n^2)$ PES length is given.

Chapter 6 examines subtraction sequences making use of *overlap graph* information. The length of subtraction sequences varies between $O(n)$ and $O(n^2)$ depending on the spatial arrangement of objects and the viewing direction. The object-space information is suitable for per-frame encoding of efficient (but not necessarily optimal) subtraction sequences.

Chapter 7 presents experimental results for a variety of CSG models using our implementation of the SCS CSG rendering algorithm, as well as Goldfeather [36, 37], Layered Goldfeather [87], Improved Layered Goldfeather [29] algorithms. The performance and capabilities of these algorithms are characterised, demonstrating the advantages and disadvantages of the SCS approach in particular circumstances.

Concluding remarks follow in Chapter 8, including discussion of some areas for potential further work.

Additional information is included as appendices — software user documentation in Appendix A and C++ source code for key algorithms in Appendix B.

Chapter 2

Constructive Solid Geometry (CSG)

Truth is ever to be found in simplicity, and not in the multiplicity and confusion of things.
— Isaac Newton

2.1 Introduction

Constructive Solid Geometry [74] (CSG) is a geometric modelling approach that forms complicated shapes from simpler geometric primitives. In contrast to surface modelling approaches both the surface and volume are represented in CSG models.

This chapter introduces the formulation of CSG models in terms of *primitives*, *operators* and *trees*. The traversal and processing of CSG models is described. In the context of hardware-based CSG rendering, tree transformation and simplification are preprocessing steps for CSG rendering algorithms such as our new algorithm *Sequenced Convex Subtraction* (SCS).

This chapter provides a background for the the chapters that follow. Chapter 3 surveys previously reported hardware-based CSG rendering techniques. The SCS algorithm is introduced in Chapter 4.

2.1.1 Primitives

The geometric shapes used in CSG modelling are referred to as *primitives*. Primitives enclose a volume, dividing space into regions inside or outside. A primitive's surface is at the boundary of the inside and outside regions of volume.

The primitives available in a CSG modelling system typically include various shapes including spheres, ellipsoids, boxes, tetrahedrons, cylinders and cones. For example, the primitives supported by the BRL-CAD modelling system are illustrated in Figure 2.1. Polygonised shapes are supported by some systems — providing they divide space into inside and outside regions unambiguously. Volumes meshed or tessellated into primitives such as tetrahedrons and boxes are suitable for CSG modelling but can result in very large CSG trees for curved surfaces.

Primitives are either bounded or unbounded — occupying a finite or infinite volume. Unbounded primitives such as planes and infinite cylinders are supported in some CSG modelling systems.

Computer graphics is often concerned primarily with surfaces of primitives since they typically form the rendered image. In CSG modelling and rendering volumes are also of central importance. CSG *operators* combine primitives into more complicated shapes based on primitive volumes, as described in the following section.

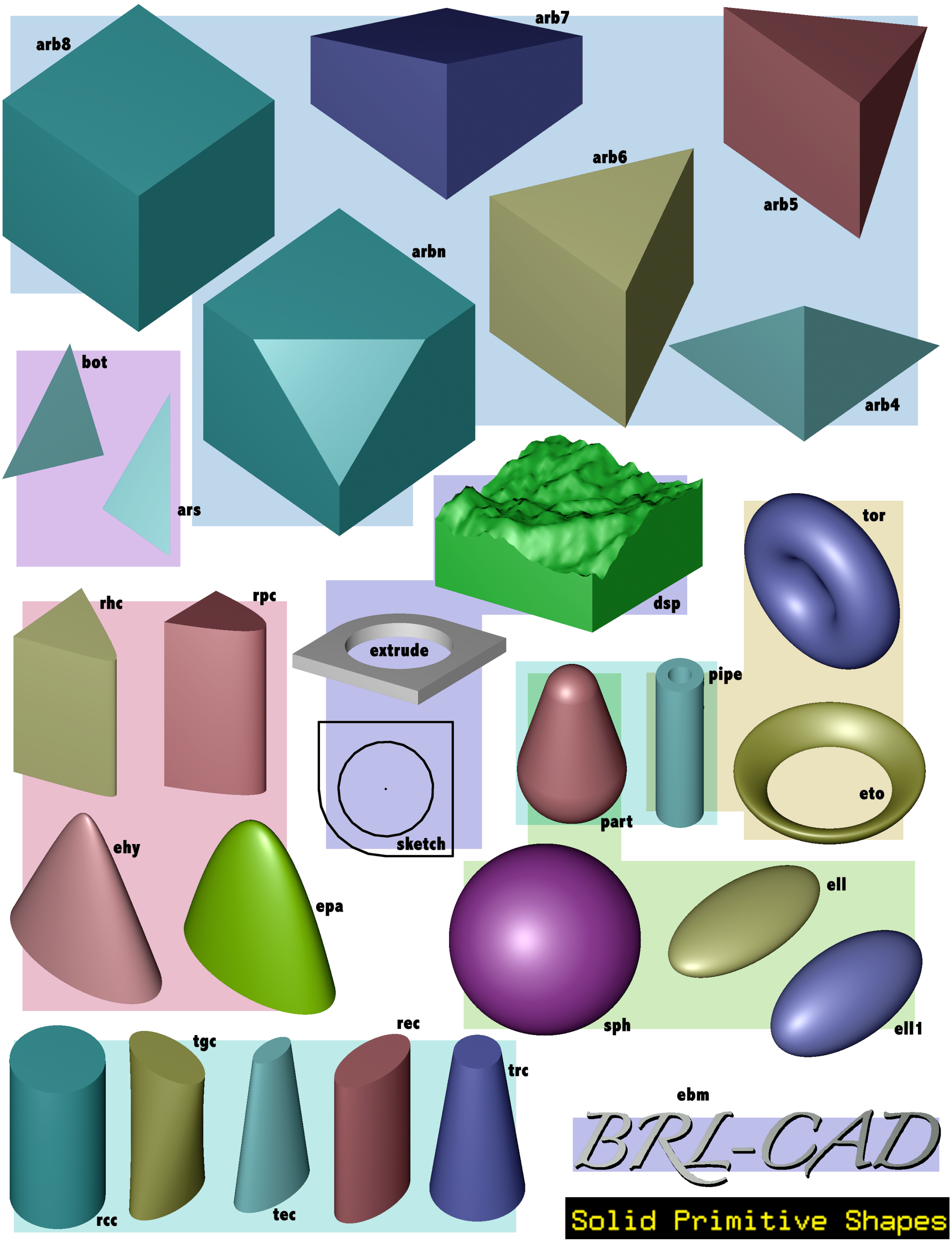


Figure 2.1: Primitives Supported by the BRL-CAD Modelling System [11].

2.1.2 Operators

Boolean algebra was developed in the 1830s by British mathematician and philosopher George Boole. As a graduate student Claude Elwood Shannon applied boolean algebra to digital circuit design[84] — and was awarded the 1940 Alfred Noble prize.

CSG primitives are volumetrically combined into more complicated shapes by the *operators* of boolean algebra. The operators are *union*, *intersection* and *difference*. This work uses the set logic notation of $\cup$, $\cap$ and $-$ for union, intersection and difference. Given primitives a and b:

- The *union*, denoted $a \cup b$ is the volume inside of a or inside of b and the surface of both a and b.
- The *intersection*, denoted $a \cap b$ is the volume inside of a and inside of b, the surface of a inside of b and the surface of b inside of a.
- The *difference*, denoted $a - b$ is the volume inside of a and outside of b, the surface of a outside of b and the surface of b inside of a.

Boolean union, intersection and difference are illustrated in Figure 2.2 (a), (b) and (c) respectively. Additional CSG operators are also sometimes used:

- The *clip* is the difference $a - b$ without the surface of b.
- The *merge* is the union $a \cup b$ without the surface of a inside b or the surface of b inside a.
- The *negation* is the volume not inside a and the surface of a.

Clip and merge are illustrated in Figure 2.2 (d) and (e) respectively.

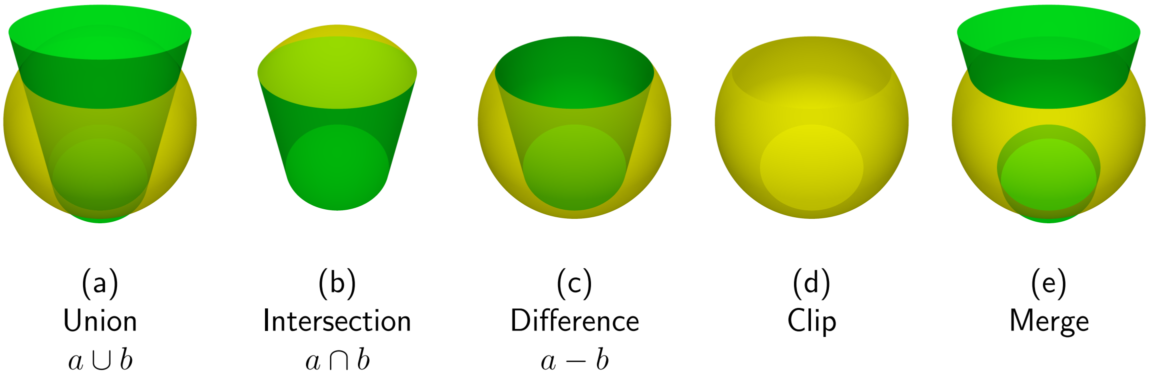


Figure 2.2: CSG Operators

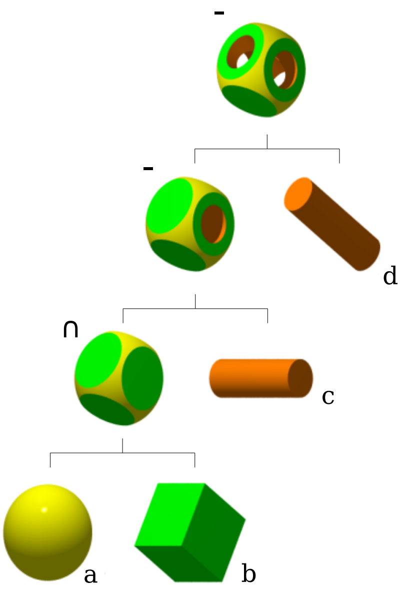


Figure 2.3: Example CSG Tree

2.1.3 Tree

Hierarchical data modelling refers to the organisation of information in a tree structure. *Object-oriented modelling* extends the tree structure to include functionality in addition to information. Hierarchical structuring of geometric models is known as *hierarchical geometric modelling*.

A *CSG tree* is a hierarchical geometric model consisting of primitive leaf nodes and boolean operator parent nodes. Children of parent nodes are either primitive or operator nodes. The *root* of the tree (the node with no parents) corresponds to the object modelled by the overall tree. Figure 2.3 illustrates a CSG model composed of a sphere, box and two cylinders: $(((a \cap b) - c) - d)$. In this example a, b, c and d are primitive (leaf) nodes and the other nodes are parents.

Hierarchical geometric models generally associate a transformation matrix with each tree node. This allows for positioning, orienting and scaling of primitives and sub-trees. By convention the transformation is relative to the parent resulting in a local coordinate system for each CSG tree node. Local coordinate systems are often desirable features for interactive editing and animation purposes.

CSG trees are typically formulated as binary trees — each parent node having two children.

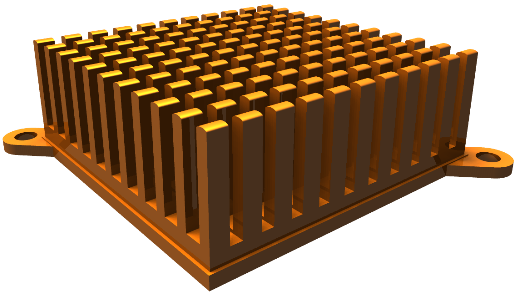


Figure 2.4: CSG Heatsink Model.

CSG model by Jonathon Haas.

Figure 2.4 is a CSG heatsink model using only box and cylinder primitives.

2.1.4 Tree Traversal

CSG algorithms generally involve CSG tree *traversal* — visiting each parent and leaf node of the tree. Traversal begins at the tree root and proceeds towards the leaves recursively:

- *Preorder traversal* visits the node before visiting the left and right trees.
- *Inorder traversal* visits the left tree, then the node, then the right tree.
- *Postorder traversal* visits the left and right trees before visiting the node.

An inorder traversal can be utilised to output the CSG expression for a given tree, from left to right. For the CSG tree in Figure 2.3 the expression is: $(((a \cap b) - c) - d)$

2.1.5 Algorithms

The focus of the work in this thesis is the specific problem of hardware-based rendering of CSG models. The CSG approach is also suitable for a variety of other solid and surface modelling computations. Some of the more common computations are mentioned in this section.

The *point classification* problem is concerned with classifying points as either inside or outside of the CSG model. Beginning at the tree root a recursive postorder tree traversal first classifies the point with respect to the children. The result at each parent node is determined by applying the boolean operator to the child classifications. The volume of a union node is the points inside any of the children. The volume of an intersection node is the points inside all of the children, and so on. In this manner point classification can be implemented for quite complicated shapes based on simple point classification methods for primitives.

Line classification divides a line into segments inside and outside of the volume, and points on the surface. It can be implemented in a similar manner to point classification by classifying lines with respect to simple primitives and applying the boolean logic of the parent node operators in a recursive postorder traversal.

Ray casting determines the first object surface intersection point in a given direction from a particular point. This query is more specific than general line classification, but can be implemented in terms of line classification.

Ray tracing is a general purpose rendering algorithm that uses ray casting extensively. For each pixel a ray is cast into the world to determine the closest point along the line on the surface of an object, if any. For detecting shadows, rays are cast from object surfaces towards light sources. Additional ray casting is also used for rendering reflection and refraction by casting rays from object surfaces further into the world.

Visibility between two points can also be implemented using ray casting.

The *shadow volume* is the infinite region of shadow cast by an object with respect to a light source. Shadow testing can be implemented by casting rays towards the light source. Another approach is shadow volume point classification [26, 31].

Collision detection is concerned with detecting the contact or overlap of multiple objects. The intersection of CSG models can be expressed and evaluated in terms of an intersection node combining a pair of CSG trees.

Boundary evaluation extracts the surface of a solid CSG model for the purpose of surface area computation, other analysis or rendering. A tessellated polygonal boundary representation is commonly used for interactive display purposes.

CSG solid models can be converted [58] to spatial data structures such as voxels, octrees, BSP trees and so on. Spatial data structures are efficient for spatial queries such as point classification and ray casting.

2.2 Tree Transformation

The arrangement of CSG trees can be changed while preserving the same volume and surface. Systematic tree transformation is often a necessary preparatory step for subsequent processing such as rendering. This section details tree transformation algorithms useful for hardware-based CSG rendering: sum-of-products, positive form and left/right heavy.

2.2.1 Sum-of-Products Form

CSG tree normalisation converts a general CSG tree to a form suitable for certain CSG algorithms. Following Hioyuki et al. [82], a *normalisation* method for converting a CSG tree to *sum-of-products* form was proposed by Goldfeather et al. [36, 37].

Normalisation flattens an arbitrarily arranged tree into a sum-of-products form that is amenable to hardware-oriented CSG rendering algorithms. Recursive traversal is generally utilised by tree related algorithms requiring memory proportional to the deepest level of recursion. Tree flattening facilitates non-recursive traversal using a fixed amount of graphics hardware storage. However, CSG tree normalisation can result in a larger overall tree [37]. Normalisation is used by CSG rendering algorithms including Trickle, Goldfeather and our new *Sequenced Convex Subtraction* (SCS) algorithm.

A collection of primitives related by boolean union is called a sum, whilst a collection related by boolean intersection or difference is called a product. CSG rendering algorithms tend to process products in an inner loop and the sum-of-products in an outer loop. A CSG tree in sum-of-products form is said to be *normalised* and has the following characteristics:

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

$$\begin{array}{rrcl}
E_1: & x-(y\cup z) & = & (x-y)-z \\
E_2: & x\cap(y\cup z) & = & (x\cap y)\cup(x\cap z) \\
E_3: & x-(y\cap z) & = & (x-y)\cup(x-z) \\
E_4: & x\cap(y\cup z) & = & (x\cap y)\cap z \\
E_5: & x-(y-z) & = & (x-y)\cup(x\cap z) \\
E_6: & x\cap(y-z) & = & (x\cap y)-z \\
E_7: & (x\cup y)-z & = & (x-z)\cup(y-z) \\
E_8: & (x\cup y)\cap z & = & (x\cap z)\cup(y\cap z)
\end{array}$$

Table 2.1: Set Equivalences for Tree Normalisation [37]

Algorithm 2.1 CSG Tree Normalisation

```
Require: $N$ is the current tree node
Require: $N_{\text{left}}$ and $N_{\text{right}}$ are the left and right subtrees of $N$
Require: $E_1, E_2, \cdots, E_8$ are the set equivalences for normalisation
  if $N$ is a parent node then
    loop
      [Using $E_1$–$E_6$ before $E_7$ or $E_8$.]
      while $N = E_{i_{\text{left}}}$ do
        $N \leftarrow E_{i_{\text{right}}}$
      normalise($N_{\text{left}}$)
      if $N = \cup$ then
        break
      if $N_{\text{right}} \neq$ parent and $N_{\text{left}} \neq \cup$ then
        break
    normalise($N_{\text{right}}$)
```

In the tree normalisation algorithm given in Algorithm 2.1 [37] the equivalence substitutions in Table 2.1 are applied recursively from the top to the bottom of the tree. The substitutions handle all possible configurations at a node of an unnormalised tree. The process may increase the overall size of the tree due to substitutions E_2, E_3, E_5, E_7 or E_8. The set equivalences in Table 2.1 are illustrated in Figure 2.5 and Figure 2.6.

Although there are other ways to normalise a CSG tree, this approach has the following characteristics [37]:

- It terminates given any CSG tree as input.
- The tree is normalised upon completion.
- Each restructuring step requires only local information.

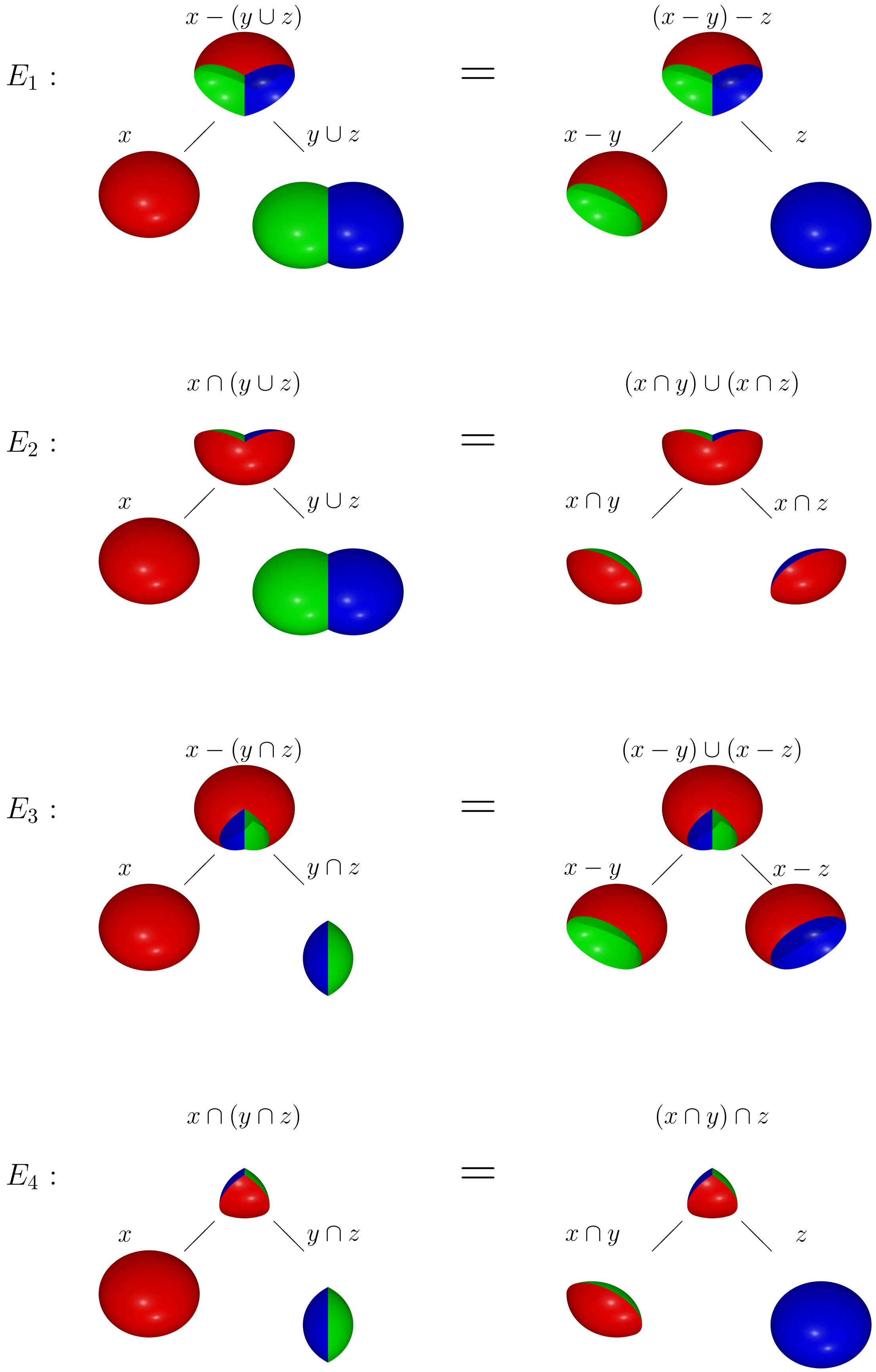


Figure 2.5: Set Equivalences 1–4 for CSG Tree Normalisation

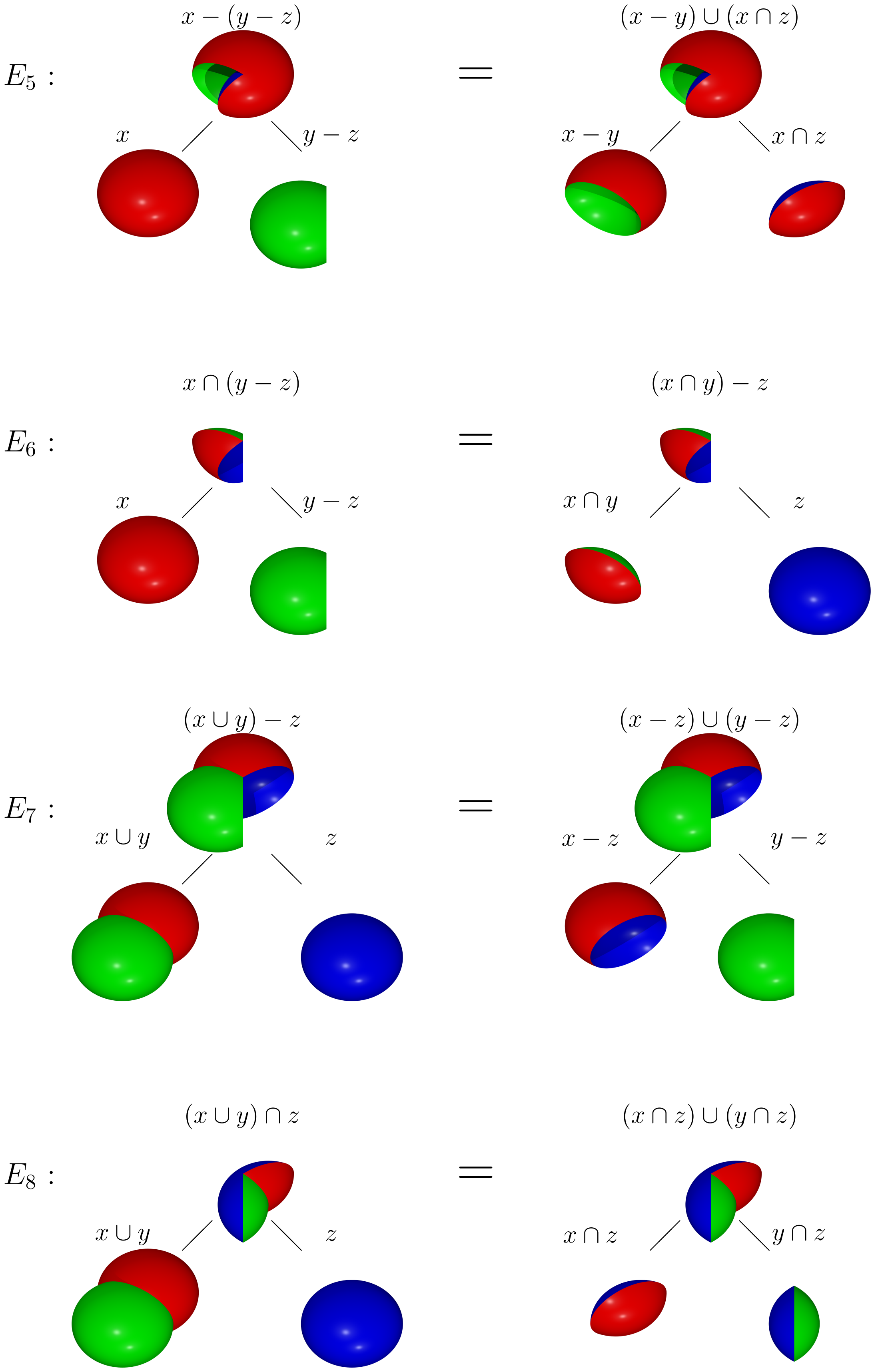


Figure 2.6: Set Equivalences 5–8 for CSG Tree Normalisation

2.2.2 Positive Form

A CSG tree can be represented in *positive form* using ∪ and ∩ operators and negation of leaf nodes. The conversion of CSG trees to positive form discussed in this section follows Rossignac and Voelcker [80].

The relationships in Table 2.2 are known as De Morgan's laws or De Morgan's theorem. Named after the 19th century logician Augustus De Morgan, they originate from classical propositional logic. Casually phrased they are (1) When neither x or y are true, both x and y are false, and (2) When x and y together are false, either x is false or y is false.

$$\begin{array}{lccc}
(1) & \overline{x \cup y} & = & \overline{x} \cap \overline{y} \\
(2) & \overline{x \cap y} & = & \overline{x} \cup \overline{y}
\end{array}$$

Table 2.2: DeMorgan's Laws [59]

The transformations listed in Table 2.3 are based on De Morgan's laws. The left and right side of each transformation and can be shown to be logically equivalent. The transformations are applied in a preorder traversal of the CSG tree — visiting the parent node, applying the relevant transformation (or none at all), then visiting the left and right children recursively.

This positive reformulation of CSG trees has been utilised for a variety of CSG algorithms [80, 77, 23, 78, 43, 46, 76] including rendering. Advantages of the positive formulation with respect to sum-of-products include notational clarity, structural similarity to general CSG tree, and avoiding sum-of-products potential exponential tree growth [77]. Positive reformulation is a critical initial step of the BList [78] and CST [46] CSG rendering algorithms.

Positive form also facilitates the *right heavy* tree formulation described in the next section.

$$\begin{array}{rcl}
x - y & \rightarrow & x \cap \overline{y} \\
\overline{x \cup y} & \rightarrow & \overline{x} \cap \overline{y} \\
\overline{x \cap y} & \rightarrow & \overline{x} \cup \overline{y} \\
\overline{\overline{x}} & \rightarrow & x
\end{array}$$

Table 2.3: Node Transformations into Positive Form [80]

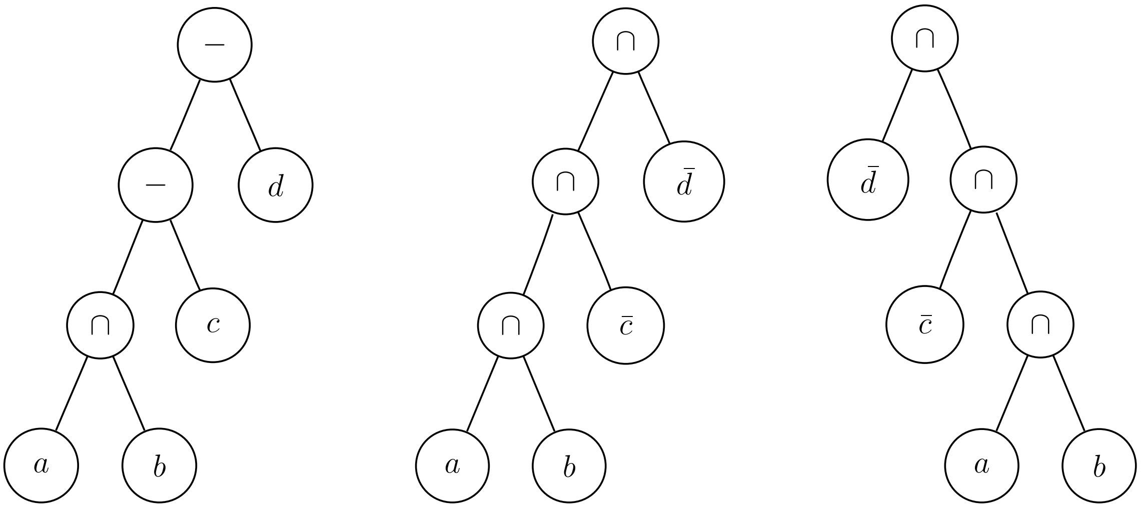
2.2.3 Left/Right Heavy

A positive form CSG tree can be reordered at each $\cup$ or $\cap$ parent node due to both $\cup$ and $\cap$ operators being commutative. That is, the order of the children of $\cup$ and $\cap$ does not affect the surface or solid represented by the CSG tree: $(a \cup b) = (b \cup a)$ and $(a \cap b) = (b \cap a)$, while $(a - b) \neq (b - a)$ since the $-$ operator is not commutative.

A CSG tree is said to be *left heavy* if the furthest descendant is via the left child of each parent node. Any positive form tree can be made left heavy by swapping children of parent nodes as necessary. Similarly, a positive form tree can be made *right heavy* by swapping children of each parent node so that the furthest descendant is via the right child of each parent node.

The *height* of a node in a CSG tree is the maximum of the heights of the child nodes, plus one. Leaf nodes have a height of one. In a postorder traversal of the CSG tree the height of each child of each parent node is determined recursively, then the children are swapped to make the subtree left or right heavy, as necessary. The traversal time is $O(n)$ with respect to the number of nodes in the CSG tree, with each node being visited once.

The CSG tree $((a \cap b) - c) - d$ (from Figure 2.3) is illustrated in Figure 2.7 in (b) left heavy positive form, and (c) right heavy positive form.



(a)
CSG Tree
$((a \cap b) - c) - d$

(b)
Left Heavy
$((a \cap b) \cap \bar{c}) \cap \bar{d}$

(c)
Right Heavy
$\bar{d} \cap (\bar{c} \cap (a \cap b))$

Figure 2.7: CSG Tree as Left and Right Heavy

2.3 Tree Simplification

2.3.1 Algebraic Tree Pruning

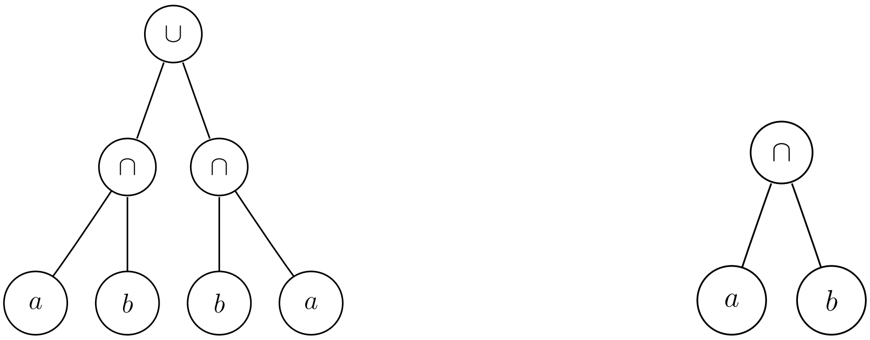
CSG trees can often be simplified by *pruning* subtrees that do not contribute to the overall geometric shape. This section describes an algebraic approach to CSG tree pruning, the next section considers geometric information for the purpose of tree pruning

Boolean algebra can be utilised to prune CSG trees independently of the geometric model. Table 2.4 lists transformations corresponding to possible parent node configurations with x representing any geometry or CSG tree. In a postorder traversal of the tree nodes matching the left side of the transformation are substituted with the right side.

$$\begin{array}{rcl}
x \cap x & \rightarrow & x \\
x \cap \overline{x} & \rightarrow & \emptyset \\
x - x & \rightarrow & \emptyset \\
x \cup x & \rightarrow & x \\
x \cap \emptyset & \rightarrow & \emptyset \\
x - \emptyset & \rightarrow & x \\
x \cup \emptyset & \rightarrow & x
\end{array}$$

Table 2.4: Algebraic Tree Pruning

In practice, thoroughly applying algebraic node pruning involves evaluating the equivalence of two CSG subtrees that may be structurally different. Figure 2.8 illustrates the simplification of a CSG tree by recognising the commutativity of the $\cap$ operator: $(a \cap b) = (b \cap a)$. The tree could be pruned further in the case that a is known to be geometrically the same as b: $x \cap x \rightarrow x$



(a) Tree $((a \cap b) \cup (b \cap a))$ in the form $x \cup x$ from Table 2.4

(b) Equivalent pruned CSG tree with the substitution $x \cup x \rightarrow x$ applied.

Figure 2.8: Algebraic CSG Tree Pruning

2.3.2 Bounding Volume Tree Pruning

Bounding volumes and *bounding volume hierarchies* are important techniques for rendering [24, 81, 96] and other problems such as collision detection. They can also be applied to CSG trees for the purpose of pruning.

Bounding volumes completely enclose geometry along with additional space called the "void area" [81]. Geometric queries such as point classification and intersection can be implemented more efficiently for geometrically simple bounding volumes than the complex geometry they enclose. Well known bounding volumes include *Axis-Aligned Bounding Boxes* (AABBs) [81], *Oriented Bounding Boxes* (OBBs), spheres [24], cylinders and convex hulls [91]. In practice the accuracy, performance and ease of implementation of bounding volume schemes vary. While bounding spheres are simple to implement they can result in large void areas for long and thin geometry. The convex hull of a complicated tessellated geometry generally minimises void area while being relatively complex and computationally intensive. The suitability of a bounding volume scheme can depend on the specific application. In practice AABBs and spheres are commonly used. The use of AABBs for CSG tree pruning has been reported previously [20, 37, 97]. For the purpose of this discussion no particular bounding volume scheme is assumed.

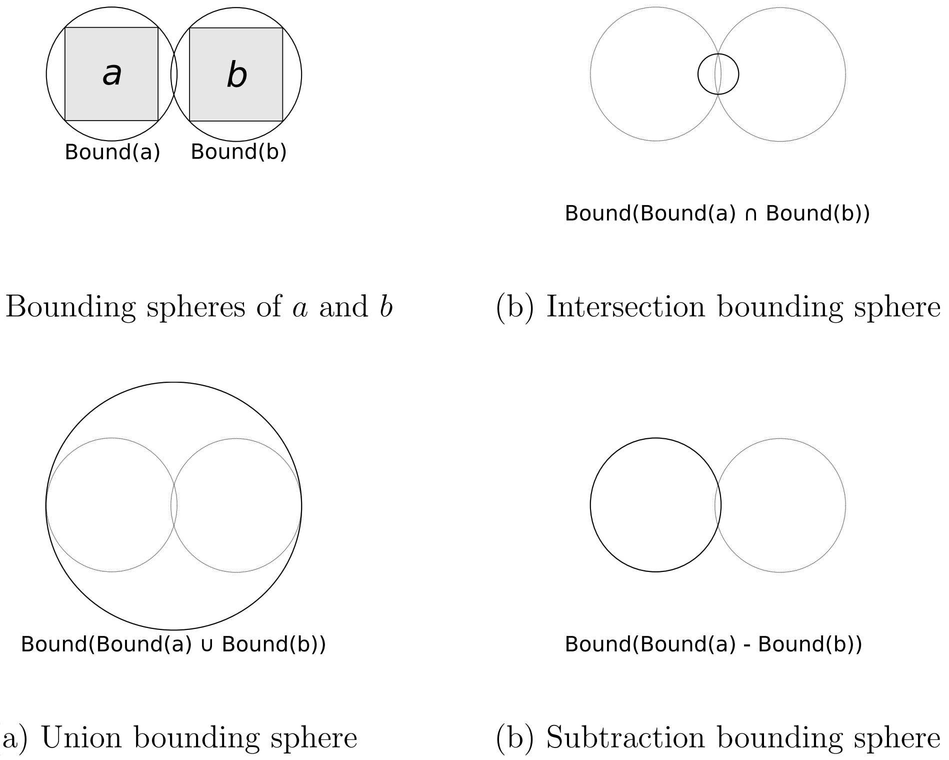
For CSG tree pruning a bounding volume is determined for each leaf and parent node in the tree, in a postorder traversal. Leaf node bounding volumes are derived directly from the geometry. Parent node bounding volumes are derived from the children according to the rules in Table 2.5.

$$\begin{array}{lcl} \mathrm{Bound}(x \cap y) & = & \mathrm{Bound}(\mathrm{Bound}(x) \cap \mathrm{Bound}(y)) \\ \mathrm{Bound}(x - y) & = & \mathrm{Bound}(\mathrm{Bound}(x) - \mathrm{Bound}(y)) \\ \mathrm{Bound}(x \cup y) & = & \mathrm{Bound}(\mathrm{Bound}(x) \cup \mathrm{Bound}(y)) \end{array}$$

Table 2.5: Parent Node Bounding Volume

CSG tree pruning is incorporated into the postorder traversal of the tree. Leaf nodes with an empty bounding volume are pruned. Intersection and subtraction parent nodes are pruned if (and only if) the intersection of the bounding volumes of the children is empty.

Parent node bounding volumes (as given in Table 2.5) can result in additional void area. Bounding volume schemes generally can not represent the intersection, union and subtraction of the volumes without introducing additional void area. For example the


a
b
Bound(a)
Bound(b)
Bound(Bound(a) ∩ Bound(b))
Bound(Bound(a) ∪ Bound(b))
Bound(Bound(a) - Bound(b))

(a) Bounding spheres of a and b

(b) Intersection bounding sphere

(a) Union bounding sphere

(b) Subtraction bounding sphere

Figure 2.9: Bounding Volume CSG Tree Pruning

intersection of two spheres is not generally spherical and would be bounded by a sphere that encloses the intersection. Nodes that could be pruned might not be pruned due to having a non-empty bounding volume that corresponds to void area introduced by the bounding volume scheme itself.

The use of sphere bounding volumes is illustrated in Figure 2.9 for (a) two geometries and their bounding spheres and the bounding spheres of (b) intersection, (c) union and (d) subtraction. Even though the geometries a and b do not overlap, the intersection and subtraction bounding volumes are non-empty due to the overlap of the bounding volume void areas. Also the union bounding volume is large in comparison to the volumes of a and b.

Other issues also arise with bounding volumes. The cumulative void area of parent nodes can is sensitive to the particular structure of the tree. In fact, void area can generally be reduced by converting the tree to sum-of-products form [62]. Unbounded geometry such as planes can not be represented by bounded volume representations such as boxes and spheres. Despite these issues and limitations bounding volumes have been applied to CSG tree pruning successfully [37, 97].

Bounding volume CSG tree pruning is analogous in principle to view frustum culling in the conventional graphics pipeline — eliminating geometry that will not form part of the final image as early as possible in the pipeline. The performance advantage stems from simple processing of bounding volumes being faster than (repeated) rasterisation of (possibly complicated) geometry.

As formulated here bounding volume pruning is a generic pre-processing step for any CSG rendering algorithm, including our new SCS algorithm. The overlap-graph subtraction sequences described in Chapter 6 incorporate bounding volume pruning without the disadvantage of cumulative volume approximation at each parent node.

2.3.3 Null Object Detection

The *Null-Object Detection* (NOD) and *Same-Object Detection* (SOD) problems are of interest in the fields of geometric modelling, CAD/CAM, robotics, computer graphics and computer vision. In the CSG context NOD determines if a tree or node is equivalent to an empty (or null) volume. SOD determines if two trees or nodes correspond to equivalent volumes. NOD and SOD take both the leaf node geometry and overall structure of the tree into consideration.

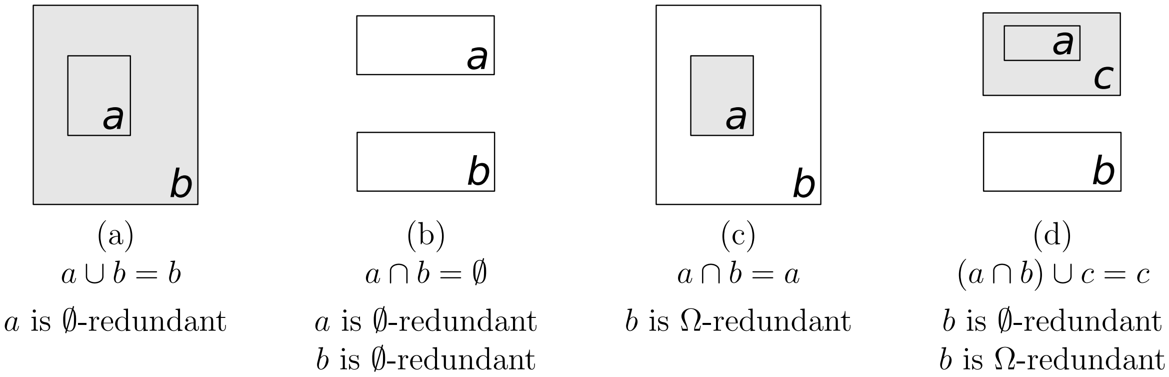
NOD is important in applications such as detecting collision between a robot arm and the environment — the intersection of the robot arm and the environment should always be empty. SOD arises in NC program verification — checking whether a manufactured object matches the original specification. The NOD and SOD problems are considered equivalent since SOD can be defined in terms NOD as follows: SOD(a, b) iff both NOD$(a-b)$ and NOD$(b-a)$.

For algebraic tree pruning (Section 2.3.1) SOD provides the means of evaluating the equivalence of two CSG trees. The subsequent section (Section 2.3.2) mentioned that one of the problems with bounding volume tree pruning is that empty trees might not always be detected due to the void area. Pruning CSG trees with NOD is potentially more aggressive than the bounding volume approach.

The approach to NOD reported by Tilove [94] is described here briefly.

A leaf or parent node x can be redundant in the context of a particular CSG tree T in two ways. Iff substituting x with the empty set $\emptyset$ in T results in the same volume as T then x is $\emptyset$-redundant. Iff substituting x with the universal set[1] Ω in T results in the same volume as T then x is Ω-redundant. Some examples of $\emptyset$ and Ω redundancy are

[1] The universal set Ω is either 2D or 3D Euclidean space: $\mathbf{E}^2$ or $\mathbf{E}^3$



(a)
$a \cup b = b$
a is $\emptyset$-redundant

(b)
$a \cap b = \emptyset$
a is $\emptyset$-redundant
b is $\emptyset$-redundant

(c)
$a \cap b = a$
b is Ω-redundant

(d)
$(a \cap b) \cup c = c$
b is $\emptyset$-redundant
b is Ω-redundant

Figure 2.10: $\emptyset$-redundancy and Ω-redundancy for NOD [80]

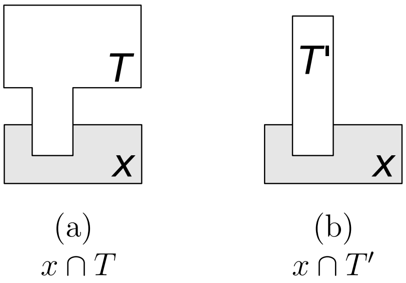
given in Figure 2.10.

In a CSG representation of the empty set all *positive* leaf nodes are $\emptyset$-redundant and all *negative* leaf nodes are Ω-redundant. Positive nodes are those whose path from the root branches to the right of a subtraction an even number of times. All other nodes are negative.

It is only necessary to check that all positive leaf nodes are $\emptyset$-redundant since negative nodes do not contribute volume to T. Leaf node $\emptyset$-redundancy testing utilises a *localised* CSG tree T' in the region of x by pruning leaf nodes not overlapping x. Geometric and bounding volume comparison is used for pairwise leaf node overlap testing. Usually T' is a much smaller tree than T due to x usually overlapping only a few other leaf nodes. The localised tree T' is equivalent to T within the region of x — as illustrated in Figure 2.11.

If $x \cap T' = \emptyset$ then x is $\emptyset$-redundant and pruned from T. Otherwise, T' is not empty and therefore T is not empty and the NOD algorithm terminates. The evaluation of $x \cap T'$ involves either classifying the geometry of x against T' or evaluating the boundary.

The approach has the advantage of avoiding full boundary evaluation by exploiting the local simplicity of mechanical parts with few overlapping pairs of geometry [94].



(a)
$x \cap T$

(b)
$x \cap T'$

Figure 2.11: NOD CSG Tree Localisation $x \cap T = x \cap T'$ [94]

2.3.4 Active Zones

Following the *Null-Object Detection* (NOD) algorithm by Tilove [94] the *Active Zone* approach was formulated by Rossignac et al. [80]. Active zones provide efficiency improvements for boundary evaluation, redundancy elimination, interference detection and rendering of CSG trees. This section introduces the active zone concept and the application of it to tree pruning.

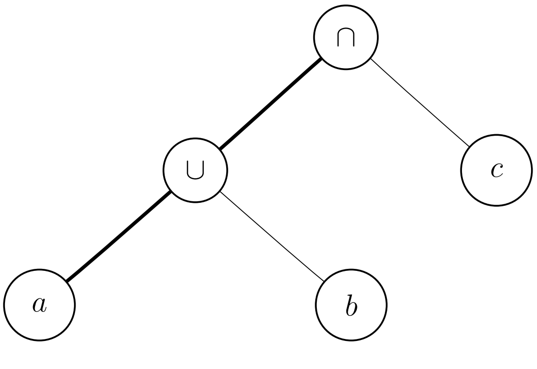
The active zone Z of a CSG tree node x is the volume in which changes to the node affect the volume and surface of the overall model. The active zone of a node depends on the geometry and structure of the CSG tree but is independent of the node itself. The redundancy of a node x is determined by checking the relationship with the active zone Z_x in the context of a particular CSG tree:

- x is $\emptyset$-redundant iff $Z_x \cap x = \emptyset$.
- x is Ω-redundant iff $Z_x - x = \emptyset$.
- x is both $\emptyset$-redundant and Ω-redundant iff $Z_x = \emptyset$.

Active zones are CSG trees defined in terms of *branching nodes* in the overall tree. In a positive-form tree (discussed previously in Section 2.2.2) branching nodes of x are the children of the path from x to the root, apart from those in the path itself.

- Branching nodes that are children of $\cap$ operators are called i-nodes
- Branching nodes that are children of $\cup$ operators are called u-nodes

An illustration of i-node and u-nodes is given in Figure 2.12 for a simple CSG tree.



Thick lines indicate the path from a to the root.
b and c are the branching nodes of a.
b is a u-node of a.
c is an i-node of a.

Figure 2.12: Active Zone i-nodes and u-nodes

The i-nodes and u-nodes form the I-zone, U-zone and active zone Z as follows:

$$I = i_1 \cap i_2 \cap \ \cdots \ \cap i_m \cap \Omega \tag{2.1}$$

$$U = u_1 \cup u_2 \cup \ \cdots \ \cup u_n \cup \emptyset \tag{2.2}$$

$$Z = I - U \tag{2.3}$$

Note that Z is unbounded for even a tree of bounded geometry in the case that there are no i-nodes. If any i-node is empty then I and Z are also empty and x is therefore redundant. Table 2.6 applies active zone tree pruning to the CSG trees from Figure 2.10.

The active zone approach to tree pruning has the performance advantages that $Z_x \cap x$ and $Z_x - x$ are simpler trees and can be both evaluated in the same step due to their similarity [80]. While the NOD approach [94] detects nodes equivalent to the empty set, active zones detect redundant nodes independently of whether the tree is equivalent to the empty set. Active zones also provide for universal set substitutions for Ω-redundant nodes such as $a \cup \Omega \rightarrow \Omega$, $a \cap \Omega \rightarrow a$ and $a - \Omega \rightarrow \emptyset$.

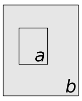
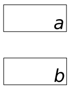
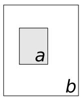
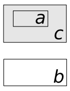
				
Tree	$a \cup b$	$a \cap b$	$a \cap b$	$(a \cap b) \cup c$
I_a	Ω	b	b	b
U_a	b	$\emptyset$	$\emptyset$	c
Z_a	$\Omega - b$	b	b	b
$Z_a \cap a$	$\emptyset$	$\emptyset$	a	$\emptyset$
$Z_a - a$	$\Omega - b$	b	$b - a$	b
a is $\emptyset$-redundant?	yes	yes	no	yes
a is Ω-redundant?	no	no	no	no
I_b	Ω	a	a	a
U_b	a	$\emptyset$	$\emptyset$	c
Z_b	$\Omega - a$	a	a	$\emptyset$
$Z_b \cap b$	$b - a$	$\emptyset$	a	$\emptyset$
$Z_b - b$	$\Omega - b$	a	$\emptyset$	$\emptyset$
b is $\emptyset$-redundant?	no	yes	no	yes
b is Ω-redundant?	no	no	yes	yes
I_c				Ω
U_c				$a \cap b$
Z_c				Ω
$Z_c \cap c$				c
$Z_c - c$				$\Omega - c$
c is $\emptyset$-redundant?				no
c is Ω-redundant?				no
Pruned Tree	b	$\emptyset$	a	c

Table 2.6: I-zones, U-zones and Active Zones for CSG Tree Pruning

2.3.5 S-bounds

The *super-bounds* (s-bounds) algorithm [22, 23] arose from robotics research as a fast approximation approach to the NOD problem. It applies bounding volumes and redundancy detection in a manner suitable for applications such as the collision detection of robot arms. This section describes the s-bounds approach and the application of them to CSG tree node pruning.

NOD determines whether an object is definitely empty, or definitely not empty. The s-bounds approach partitions space into definitely empty regions, and possibly empty regions. When the entire space is definitely empty NOD is complete. Otherwise regions of possible emptiness need additional processing using non-approximating algorithms such as active zones. Overall processing time is thereby minimised by limiting expensive detailed analysis to specific localised regions.

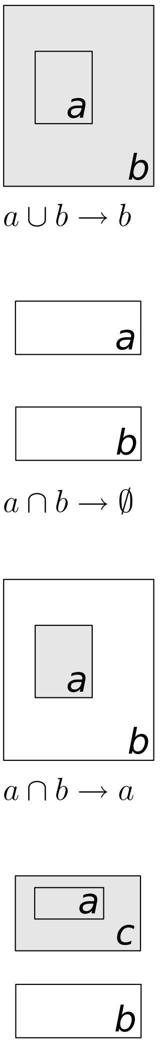
S-bounds assigns a bounding volume to each node in the tree. The AABB bounding volume scheme is used here for the purpose of explanation, although others could also be used [22, 23]. The initial bounding box for leaf nodes is derived from the geometry. The initial bounding box for parent nodes is the entire space Ω. In the initial pass of the algorithm parent bounds are derived from their children based on the parent operator (as described in Section 2.3.2) in a postorder traversal. This is called the *upwards* pass. The next pass is a preorder traversal replacing each child bound with the intersection of the parent and child bounds. This is called the *downwards* pass. The algorithm continues with upwards and downwards passes until the bounding volumes converge — remaining unchanged from one pass to the next. Eventual convergence is ensured by the monotonic decrease of bounding volumes due to the intersection applied from parents to children in the downwards pass. AABB s-bounds converge in $O(n^2)$ time with respect to the number of nodes, but more rapidly in practice [22].

Once converged the tree is inspected for empty bounding volumes. Nodes with empty bounds are pruned. If the root bound is empty the entire tree is null and processing is complete. Otherwise the remaining bounding volumes can be further processed for the purpose of redundancy or null object detection.

One advantage of the s-bound approach is that bounding volumes such as AABBs can be processed more efficiently than the actual geometry. The s-bound algorithm can be useful for CSG tree pruning and other applications even without the use of time consuming geometric processing. S-bounds can also be utilised in other applications such as boundary evaluation to minimise the regions processed in a detailed manner.

Some observations about the relationship of s-bounds to active zones (Section 2.3.4) were reported subsequently [23]. S-bounds can be sensitive to the structure of the CSG tree as well as the bounding scheme. Exact s-bounds using the exact geometry (rather than superset bounding volumes) converge after one upwards and one downwards pass. In general: s-bound(x) $\supseteq$ ($Z_x \cap x$).

Figure 2.13 applies s-bounds to the CSG trees from Figure 2.10, resulting the same tree simplifications as active zones in Table 2.6.



$a \cup b \rightarrow b$

node	init	↑	↓
$\cup$	Ω	b	b
a	a	a	a
b	b	b	b

$a \cap b \rightarrow \emptyset$

node	init	↑	↓	↑
$\cap$	Ω	$\emptyset$	$\emptyset$	$\emptyset$
a	a	a	$\emptyset$	$\emptyset$
b	b	b	$\emptyset$	$\emptyset$

$a \cap b \rightarrow a$

node	init	↑	↓	↑
$\cap$	Ω	a	a	a
a	a	a	a	a
b	b	b	a	a

$(a \cap b) \cup c \rightarrow c$

node	init	↑	↓	↑
$\cup$	Ω	c	c	c
$\cap$	Ω	$\emptyset$	$\emptyset$	$\emptyset$
a	a	a	$\emptyset$	$\emptyset$
b	b	b	$\emptyset$	$\emptyset$
c	c	c	c	c

Figure 2.13: S-bounds for CSG Tree Pruning

2.4 Discussion

This chapter presented the CSG approach for geometric modelling. Primitives are combined by means of operators arranged in a CSG tree. Tree restructuring methods preserve the volume and surface of the model while satisfying structural properties. Tree pruning can be applied to reduce the size of the CSG tree and maximise the performance of subsequent processing such as rendering.

In the context of CSG modelling processing follows the sequence:

- *Modelling* is the process of specifying primitives, operators and spatial arrangement of a geometric model.
- *Transformation* converts the CSG tree into form suitable for pruning.
- *Pruning* removes unnecessary parts of the CSG tree in order to maximise rendering performance.
- *Rendering* displays an image of the CSG model.

The CSG rendering step is our focus in this thesis. Chapter 3 reviews previously reported hardware-based rendering CSG methods and graphics hardware implementation using OpenGL. Chapters 4 and 6 detail a new algorithm for CSG rendering we call SCS.

Chapter 3

Hardware-Based CSG Rendering

Integrated circuits will lead to such wonders as home computers.

— Gordon E. Moore, 1965

3.1 Introduction

This chapter begins with a brief discussion of CPU and graphics hardware aspects of PC computers. The functionality and terminology of the OpenGL interface and related graphics hardware based algorithms for CSG rendering are then introduced. The practical application of CSG rendering algorithms is closely related to the limitations and performance characteristics of graphics hardware. Subsequent chapters in this work focus on a new approach to CSG rendering we call *Sequenced Convex Subtraction* (SCS). The new algorithms involved in SCS draw ideas and inspiration from previous techniques including the Goldfeather and Trickle algorithms, layer extraction, layer peeling and depth complexity sampling. This chapter focuses on foundational concepts in support of the chapters that follow.

3.2 Hardware

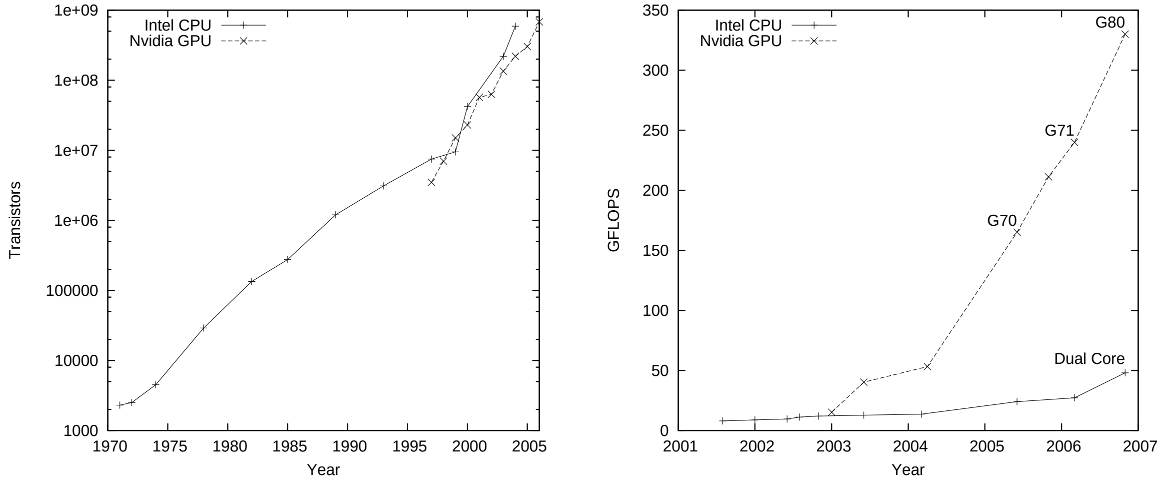
3.2.1 Performance Trends

The co-founder of Intel, Gordon E. Moore made the empirical observation in 1965 that the number of transistors on an integrated circuit roughly doubled every 12 months [65, 9, 95]. In the subsequent forty years the semiconductor industry has achieved exponential growth in transistor count, the trend widely referred to as *Moore's Law*.

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. [65]

The number of transistors in Intel *Central Processing Unit* (CPU) chips and Nvidia *Graphics Processing Unit* (GPU) chips is shown in Figure 3.1(a). Since their introduction in the mid 1990's, Nvidia graphics chip complexity has grown in a similar manner to Intel CPUs — broadly following Moore's Law.

The *Floating Point Operations Per Second* (FLOPS) is a more specific measure of the computational capacity of chips intended for processing numerical data. The FLOPS for Intel and Nvidia chips is shown in Figure 3.1(b), illustrating an approximately six-fold floating point advantage for the G80 Nvidia GPU versus an Intel 3.0GHz Core 2 Duo CPU.



(a) Transistor Count [9]

(b) Floating Point Operations Per Second [70, 71]

Figure 3.1: Intel CPU and Nvidia GPU Performance Trends

This notable advantage of graphics hardware is due to the computational paradigm of interactive graphics — pixel colours are computed individually, independently and completely for each frame. This allows for a higher latency "many at the same time" stream-based arrangement, compared to a lower latency "one at a time" general purpose CPU architecture.

A question that arises with the computational capacity offered by graphics hardware is that of adapting algorithms for CPUs to the graphics pipeline paradigm. There is currently a great deal of research interest in general purpose processing using GPUs (known as GPGPU [10]) for Fast Fourier Transform, linear system solving, motion planning, collision detection, database, sorting and other computationally demanding problems.

This work also draws on the motivation to utilise graphics hardware computational capacity to solve a problem that would be traditionally solved using a CPU-based algorithm.

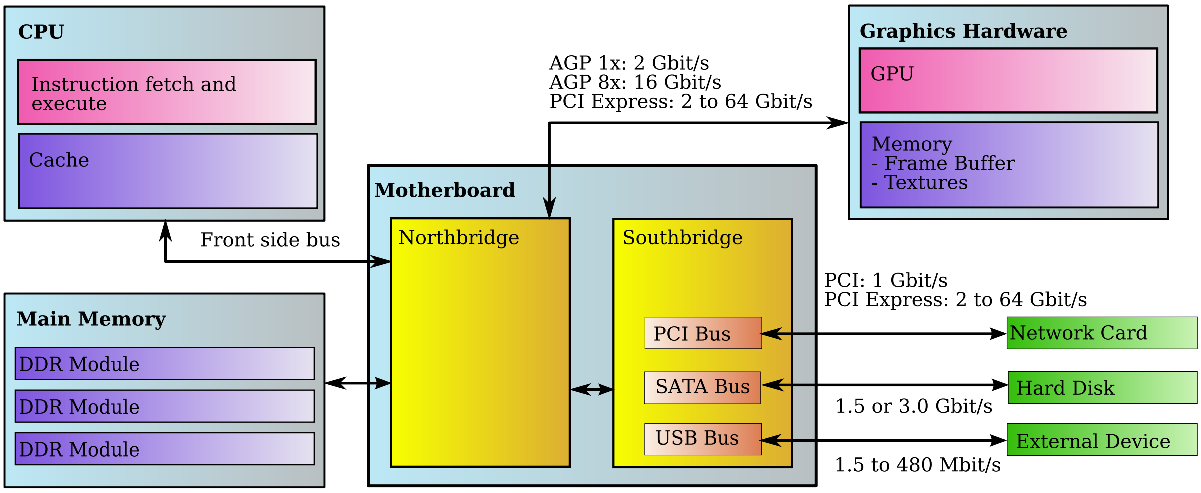


Figure 3.2: PC Architecture

3.2.2 PC Architecture

A schematic of the central components of the modern PC is given in Figure 3.2 including CPU, main memory, motherboard components, system buses and graphics hardware. One of the features of the platform is the modular hardware design, allowing for upgrading of individual components. Over time the PC platform has evolved to take advantage of available technologies, while mostly retaining the same overall arrangement. One notable part of this evolution was the introduction of the *Accelerated Graphics Port* (AGP) in 1997, providing a dedicated connection between the CPU and graphics hardware via the Northbridge. Graphics hardware had previously been attached less directly to the CPU, situated with other miscellaneous modules such as network adaptors and mass storage devices on the PCI bus. AGP also provided the capability for graphics hardware to read from main memory using the *Graphics Address Remapping Table* (GART), rather than needing to copy data across the PCI bus. One significant limitation of AGP was bandwidth asymmetry — data transfer from the graphics hardware to main memory was significantly slower than transfer from main memory to the graphics hardware. AGP was superceded by PCI Express (PCIe) in 2003.

The algorithms in this work were developed and reported in the context of AGP based PC graphics hardware.

3.3 OpenGL Graphics API

The OpenGL real-time 2D and 3D graphics *Application Programming Interface* (API) was introduced in 1992 as an open, vendor neutral, multi-platform and scalable basis for interactive graphics application development [1, 18, 19]. Based on SGI's IRIS GL library, the OpenGL standard was managed and evolved by the OpenGL *Architecture Review Board* (ARB). In 2006 control of the OpenGL API was transferred to the Khronos Group, a member-funded industry consortium focused on the creation of open standards [2]. Current members of the Khronos Group include Nvidia, ATI, Apple, Intel, Sony, Nokia, Motorola, Mitsubishi Electric, Philips, SGI and Sun Microsystems.

OpenGL is a low level interface providing access to the rasterisation, transformation, lighting, texturing and other capabilities of dedicated graphics hardware. Software implementations of OpenGL such as the Mesa 3D library [4] and the original SGI reference implementation [3] are also available.

The OpenGL specification evolves through two mechanisms. The latest hardware features are made accessible via vendor specific extensions. Widely supported extensions are then considered for inclusion in the OpenGL standard. A significant reorganisation of OpenGL happened while this work was in progress, in the form of OpenGL 2.0 [15] and 2.1. This work is concerned with standard OpenGL functionality in OpenGL 1.

3.3.1 OpenGL Graphics Pipeline

The graphics pipeline for interactive 3D graphics is composed of several processing steps as information moves from the application to the frame buffer for display. These are broadly grouped into the *transformation and lighting*, *rasterisation* and *fragment testing* stages, illustrated in Figure 3.3. The OpenGL API provides an interface for configuring the behaviour of each stage of the pipeline for achieving the desired image. The colour buffer information is used by the graphics hardware to provide the signal for the display device.

The transformation and lighting stage applies modelling and projection transformations to point, line and polygon data passed by the application. Clipping is applied to the transformed geometry, preventing further processing of geometry that is outside of the current view volume. The view volume is constrained by the edges of the view window and the near and far clipping planes at z_{near} and z_{far}. Lighting calculations are applied to each vertex of each polygon.

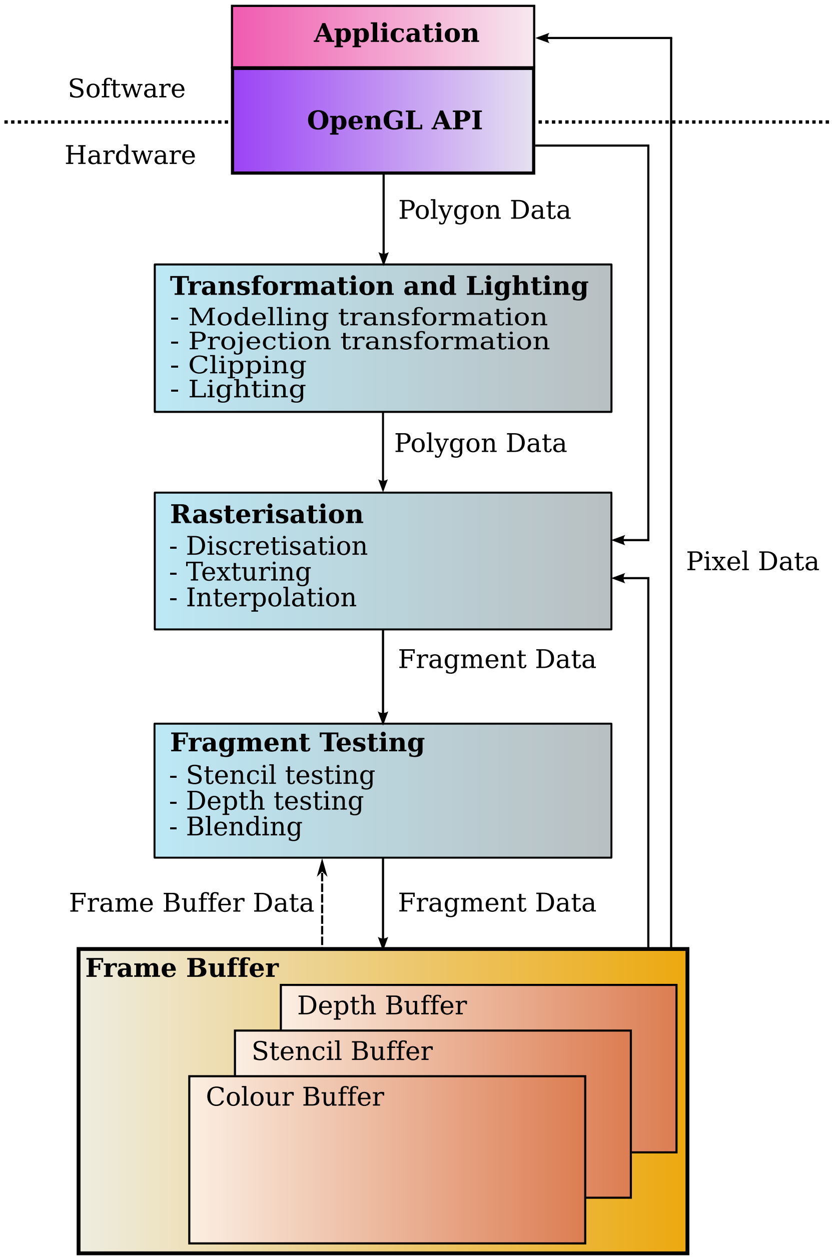


Figure 3.3: OpenGL Graphics Pipeline

The rasterisation stage converts point, line and polygon data into *fragments* that correspond to pixels in the frame buffer. Texturing and colour interpolation is applied at this stage, according to the current material, lighting and texturing OpenGL state. Fragments produced by rasterisation carry transparency and depth information in addition to colour.

The fragment testing stage applies stencil, depth and alpha tests to each fragment produced by the rasteriser. Fragments that pass these tests are written into the frame buffer. Updates to the stencil, depth and colour buffers are also individually controlled via the OpenGL buffer masking interface.

The OpenGL graphics pipeline also includes paths for drawing, reading and copying pixel data. These pixel data paths bypass certain stages relevant to polygon data processing, as further discussed in Section 3.3.5.

The following subsections focus on the depth and stencil testing stages of the graphics pipeline. These pixel-level operations provide the basis for hardware based CSG rendering in OpenGL. Foundational depth and stencil based algorithms including depth complexity sampling, layer extraction and layer peeling are described in subsequent subsections, followed by CSG rendering algorithms in Section 3.5.

3.3.2 OpenGL Depth Buffer

One of the fundamental problems in computer graphics is the visible surface problem [92]: given a collection of geometry, which parts of which shapes are visible from a particular viewing location? Parts of shapes may not be visible to the viewer because they are not facing the viewer, because the viewer is not facing the object, or because another shape is occluding the line of sight between the viewer and the object.

The primary purpose of the z-buffer algorithm is solution of the visible surface problem. The distance from the viewer is calculated for each pixel for each geometric shape as it is drawn. Since closer shapes occlude those more distant, the distance of fragments corresponding to each shape are compared to the values stored in the z-buffer. Fragments are drawn and the z-buffer updated if they are closer than the z-buffer, and ignored when their distance is greater.

The z-buffer algorithm is simple and amenable to hardware implementation. The visible surface problem is solved for each pixel independently, making the algorithm amenable to parallel implementation. The result of the z-buffer algorithm is independent of the order in which shapes are rendered. The decreasing cost of memory for z-buffer storage has made the hardware z-buffer implementation the predominant technology for visible surface determination in real-time and interactive computer graphics.

The z-buffer algorithm is an image-space approach to the visible surface problem. Object-space approaches are also well known, such as rendering sorted geometry from back to front. Ivan Sutherland contrasted ten different visible surface algorithms in his seminal paper in 1974 [92]. Object-space algorithms are still in use for a variety of applications, often in combination with z-buffering in hybrid approaches.

The OpenGL API for depth testing is summarised in Table 3.1. The function `glClearDepth` specifies the depth value to be used when the depth buffer is cleared, using the normalised range of $[0, 1]$. The `glClear` function is used to clear buffers, including the depth buffer. The `glEnable` and `glDisable` functions may be used to enable or disable depth testing. The `glDepthMask` function specifies whether the depth buffer is writable. The `glDepthFunc` function specifies the depth comparison function. Some of the available depth comparisons include `GL_NEVER`, `GL_ALWAYS`, `GL_LESS`, `GL_GREATER`, and `GL_EQUAL`.

```
glClearDepth(1.0);              // Set clear depth value
glClear(GL_DEPTH_BUFFER_BIT);   // Clear depth buffer
glEnable(GL_DEPTH_TEST);        // Enable depth testing
glDisable(GL_DEPTH_TEST);       // Disable depth testing
glDepthMask(GL_TRUE);           // Enable depth buffer modification
glDepthFunc(GL_LESS);           // Set depth test function to less-than
```

Table 3.1: OpenGL Depth Testing Settings

3.3.3 OpenGL Stencil Buffer

OpenGL supports a *stencil buffer* in addition to the depth buffer. The stencil buffer consists of at least one bit-plane, but is usually eight bit-planes. The stencil buffer is used for masking fragments according to a *stencil test* that controls which fragments update the frame buffer, in a manner similar to the depth test.

The OpenGL API for stencil testing is summarised in Table 3.2. The stencil clear value is specified by the `glClearStencil` function. The `glClear` function is used to clear buffers, including the stencil buffer. The `glEnable` and `glDisable` functions may be used to enable or disable stencil testing. The `glStencilMask` function specifies which bit-planes of the stencil buffer are updated. `glStencilFunc` specifies the stencil testing function such as `GL_EQUAL` or `GL_LESS`, a reference value and a bitwise mask. The mask allows specific stencil bits to be used or ignored during stencil testing. The `glStencilOp` function specifies the stencil operation to be applied in the following circumstances: stencil and depth test both fail; stencil test succeeds, but depth test fails; stencil and depth test both succeed.

The state of the stencil buffer is determined by the initial stencil buffer value and the subsequent result of depth and stencil testing of rasterised fragments.

```
glClearStencil(0);                  // Set stencil clear value
glClear(GL_STENCIL_BUFFER_BIT);     // Clear stencil buffer
glEnable(GL_STENCIL_TEST);          // Enable stencil testing
glDisable(GL_STENCIL_TEST);         // Disable stencil testing
glStencilMask(GL_TRUE);             // Enable stencil buffer updates
glStencilFunc(GL_EQUAL,1,1);        // Set stencil test function
glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP); // Set stencil update operation
```

Table 3.2: OpenGL Stencil Testing Settings

3.3.4 OpenGL Face Culling

Once polygons have been clipped and transformed, OpenGL provides the option to cull front or back facing polygons according to the winding order. Polygons facing away from the viewer are often not visible in the rendered image, being obscured by closer front facing polygons. Rasterising only front-facing polygons has the advantage of reducing the number of polygons entering the rasterisation stage and consequently, the number of fragments that need to be processed.

Classification of front and back polygons is based on the clockwise or anti-clockwise winding order of transformed polygon vertices. Face culling depends on polygon vertices being consistently ordered was they are passed to OpenGL. Neighbouring polygons sharing an edge need their vertices ordered so that the edge is traversed in opposite directions.

Face culling is conventionally used to avoid rasterisation of back-facing polygons of closed, consistently oriented polygonised objects. In the context of CSG rendering face culling provides a useful mechanism for separately rasterising the front and back layers of polygonised objects. Each layer is processed in a separate pass of a multiple-pass CSG rendering algorithm

The OpenGL API for face culling testing is summarised in Table 3.3. The `glEnable` and `glDisable` functions are used to enable or disable face culling. The `glCullFace` function specifies culling of either front or back faces. `glFrontFace` specifies either counter-clockwise or clockwise winding order of front faces.

```
glEnable(GL_CULL_FACE);    // Enable face culling
glDisable(GL_CULL_FACE);   // Disable face culling
glCullFace(GL_BACK);       // Specify culling of back faces
glCullFace(GL_FRONT);      // Specify culling of front faces
glFrontFace(GL_CCW);       // Specify counter-clockwise winding order of front faces
glFrontFace(GL_CW);        // Specify clockwise winding order of front faces
```

Table 3.3: OpenGL Face Culling Settings

3.3.5 OpenGL Pixel Transfer

The OpenGL graphics pipeline is primarily concerned with the processing of polygons, rasterisation of polygons into fragments, and updating the frame buffer fragment-by-fragment as illustrated previously in Figure 3.3.

OpenGL also supports pixel data processing via `glDrawPixels`, `glReadPixels` and `glCopyPixels`. Polygon processing is generally bypassed for pixel data. Rasterisation and fragment processing is applied to pixel data in the same manner as rasterised polygons. Figure 3.3 illustrates the `glDrawPixels` path from OpenGL to the rasterisation stage, the `glReadPixels` path from the frame buffer to the application, and the `glCopyPixels` path from the frame buffer to the rasterisation stage.

Pixel data is written from memory into the frame buffer using `glDrawPixels`. A variety of *formats* of pixel data are supported including stencil, depth, colour and alpha. Various data *types* of pixel data are supported such as bitmap, integer and floating point. The frame buffer is updated relative to the current *raster position*. The arrangement of pixel data in memory is specified via `glPixelStore`. Scaling and bias of pixel values is specified via `glPixelTransfer`. Pixel data can also be zoomed using `glPixelZoom`.

Pixel data is written from the frame buffer into memory using `glReadPixels`. Various pixel formats and types can be specified as for `glDrawPixels`. Processing of pixel data is specified via `glPixelStore` and `glPixelTransfer`.

Pixel data can be copied within the frame buffer using `glCopyPixels`. Either colour, depth or stencil values are copied relative to the current raster position. Processing of pixel data is specified via `glPixelStore`, `glPixelTransfer` and `glPixelZoom`.

Pixel transfer functionality is useful in the context of CSG rendering in several ways. `glReadPixels` can be used to examine frame buffer stencil buffer values. In the next section (Section 3.4.1) this approach is used to measure depth complexity — useful information for CSG rendering. Multiple depth buffers can be emulated by copying pixel data in and out of the frame buffer using `glDrawPixels` and `glReadPixels`. Performance issues related to OpenGL pixel transfer are examined in Section 7.2.

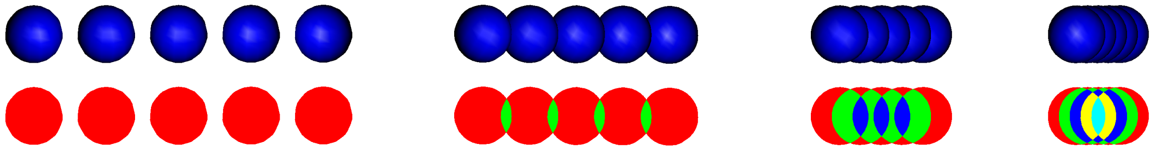
```
glDrawPixels(w,h,GL_RGB,GL_UNSIGNED_BYTE,pixels);       // Draw RGB pixel data
glReadPixels(x,y,w,h,GL_RGB,GL_UNSIGNED_BYTE,pixels);   // Read RGB pixel data
glCopyPixels(x,y,w,h,GL_DEPTH);                         // Copy depth data
```

Table 3.4: OpenGL Pixel Transfer

3.4 Frame Buffer Operations

The previous section described important aspects of the OpenGL graphics pipeline for CSG rendering applications — depth and stencil buffers, face culling and pixel transfer. This section describes the utilisation of these OpenGL functionalities for CSG rendering purposes.

3.4.1 Depth Complexity



(a) $k = 1$ (b) $k = 2$ (c) $k = 3$ (d) $k = 5$

Figure 3.4: Depth Complexity from Different Viewing Directions

The *depth complexity* of a pixel is the number of fragments drawn to the pixel [92]. In this work we refer to the maximum pixel depth complexity as the depth complexity of the scene, denoted k, which varies with the arrangement of objects and the viewing direction. Figure 3.4 illustrates increasing depth complexity of a row of spheres as viewed from a direction increasingly along the row. Red is used to illustrate depth complexity of one, green for two, blue for three, yellow for four and cyan for five.

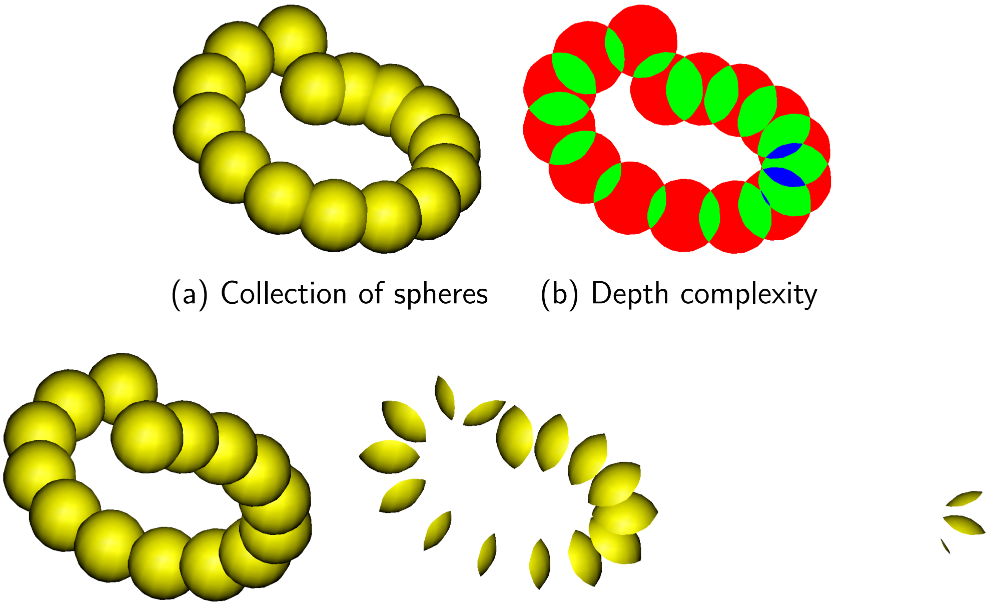
The OpenGL stencil test can be used to measure depth complexity for a particular viewing direction. In Algorithm 3.1 the stencil test is configured to increment the stencil for each fragment rendered. The depth test is not used, and the colour buffer is not updated. Once all fragments have been processed, each stencil value contains the number of fragments rasterised at that pixel. The maximum stencil value can be found by examining the stencil buffer after reading into system memory.

Algorithm 3.1 Depth Complexity Sampling

```
for all pixels do
    $stencil \leftarrow 0$

for all fragments do
    $stencil \leftarrow stencil + 1$

$k \leftarrow 0$
for all pixels do
    if $stencil > k$ then
        $k \leftarrow stencil$
```

One potential issue with depth complexity sampling is integer overflow when the depth complexity exceeds the maximum value of the stencil buffer. A depth complexity of up to 255 can be directly measured using an eight bit stencil buffer, enough for many applications. Larger depth complexities could be measured by testing groups of $2^n - 1$ objects, where n is the number of stencil buffer bits. The stencil buffer of each group would need to be read into memory and combined by taking the sum.

3.4.2 Layer Extraction



(a) Collection of spheres (b) Depth complexity

(i) 1st layer (ii) 2nd layer (iii) 3rd layer

Figure 3.5: Layer Extraction of Sphere Front Faces

An image-space technique closely related to depth complexity measurement is *layer extraction* [87]. A *layer* is a z-buffer representation of a set of geometric surfaces, with the limitation that only one surface can be stored at each pixel. The number of layers necessary to represent a collection of surfaces is the depth complexity — the maximum number of surfaces rasterised at any pixel. Hardware based algorithms can process arbitrary collections of surfaces by extracting and processing each layer in turn.

Layer extraction utilises the stencil buffer by constraining rendering to the i^{th} fragment at each pixel, where $1 \leq i \leq k$ and k is the depth complexity. The first layer is the set of fragments first rasterised at each pixel. The second layer is the second set of fragments rasterised at each pixel, and so on. In the layer extraction algorithm (Algorithm 3.2), the stencil is incremented at each pixel for each fragment. The pixel depth is only updated when the stencil equals the index of the desired layer.

Layers are used in CSG rendering for processing concave geometry in the frame buffer. Since the z-buffer can only store one surface at each pixel, it is necessary to extract and process each layer of concave shapes individually. Figure 3.5 illustrates the three layers formed by the front facing surfaces of a collection of spheres.

Algorithm 3.2 Layer Extraction

```
Require: $i \in [1, k]$, where $k$ is the scene depth complexity
  for all pixels do
    $stencil \leftarrow 0$
  for all fragments $F$ do
    $stencil \leftarrow stencil + 1$
    if $stencil = i$ then
      $z \leftarrow z_F$
```

3.4.3 Layer Peeling

A variation of layer extraction called *layer peeling* is useful for problems such as the correct rendering of transparent surfaces and CSG rendering. While layer extraction is not concerned with the depth relationship of fragments in different layers, peeled layers are sorted in front-to-back or back-to-front order with respect to z. Peeling is more difficult to implement in OpenGL than layer extraction.

The front-to-back layer peeling algorithm is presented in Algorithm 3.3. Layers are peeled iteratively from front to back, using the previous layer as the basis for determining the next. The algorithm finds the closest surface which is further than the current layer surface. Layer peeling requires two depth buffers: one to store the previous layer and another to determine the next layer. Fragments are depth tested against the two z-buffers to determine the next layer surface. The first depth test rejects fragments closer than the previous layer. The second depth test resolves the nearest fragment of those passing the first depth test. Once a peeling operation is completed, the buffer storing the previous layer is no longer needed.

In Algorithm 3.3 the stencil buffer is used to maintain a per-pixel collision count. This resolves the problem of multiple fragments having the same depth at the same pixel. Without maintaining a collision count coincident fragments would collapse into the same layer. In CSG rendering applications it is important that each fragment belongs to a layer, and that each pixel in a layer corresponds to one fragment. Using closed geometry the number of front-facing layers at a pixel should be the same as the number of back-facing layers. Front and back-facing layer pairs define a volumetric layer, corresponding to the enclosed volume. In this manner surfaces in the form of fragments can be tested against volumes in the form of pairs of layers.

Back-to-front layer peeling is performed in the same manner, with some reconfiguration to Algorithm 3.3. In the first pass, z_{next} is initialised to z_{far}. To peel from back to front, the depth test is reversed to $z_{\text{next}} < z_F < z_{\text{prev}}$.

Algorithm 3.3 Front to Back Layer Peeling

```
Require: $z_{\text{prev}}$ is the previous layer
Require: $z_{\text{next}}$ is the next layer
Require: $z_{\text{prev}} \leftarrow z_{\text{near}}$ in the first pass
Require: $\textit{stencil} \leftarrow 0$ in the first pass
  [Decrement collision count]
  for all pixels do
    $z_{\text{next}} \leftarrow z_{\text{far}}$
    if $\textit{stencil} > 0$ then
      $\textit{stencil} \leftarrow \textit{stencil} - 1$
  [Determine next layer]
  for all fragments $F$ do
    if $z_{\text{prev}} < z_F < z_{\text{next}}$ then
      if $\textit{stencil} = 0$ then
        $z_{\text{next}} \leftarrow z_F$
  [Flag pixels needing collision count update]
  for all pixels do
    if $\textit{stencil} = 0$ then
      $flagPixel \leftarrow true$
    else
      $flagPixel \leftarrow false$
  [Compute collision count for flagged pixels]
  for all fragments $F$ do
    if $flagPixel$ and $z_F = z_{\text{next}}$ then
      $\textit{stencil} \leftarrow \textit{stencil} + 1$
```

Implementation of layer peeling in OpenGL is difficult[1] due to the requirement of two depth buffers and two depth tests per fragment. Layer peeling can be implemented in OpenGL, with a single depth buffer but is considered too slow for practical use. Architectures supporting deep frame-buffers, multiple z-buffers, extended fragment testing or high-bandwidth are more promising from a performance point of view [32, 63, 64].

3.4.4 Parity Clipping

A layer in the z-buffer can be clipped against a closed object by counting the number of fragments in front of (or behind) each pixel. An odd number indicates a pixel inside the volume of the object, while an even number indicates a pixel outside the volume of the object.

In an intersection operation odd parity pixels are preserved and even parity pixels are reset. In a subtraction operation even parity pixels are preserved and odd parity pixels are reset.

Parity clipping can be implemented in OpenGL by toggling a stencil bit [18] at each pixel for every fragment closer than the current depth value. The parity clipping algorithm is presented in Algorithm 3.4.

Algorithm 3.4 Parity Clipping

```
Require: $z_O$ is the z-buffer surface being clipped
Require: $C$ is the convex or concave object being clipped against
  [Clear stencil buffer]
  for all pixels do
    $stencil \leftarrow 0$
  [Count fragments in front of z-buffer at each pixel]
  for all fragments of $C$ do
    if $z_C < z_O$ then
      $stencil \leftarrow stencil + 1$
  [Clip pixels with even or odd parity]
  for all pixels do
    if $C$ subtracted then
      if $stencil$ is odd then
        $z_O \leftarrow z_{\text{far}}$
    if $C$ intersected then
      if $stencil$ is even then
        $z_O \leftarrow z_{\text{far}}$
```

[1] Layer peeling has reportedly been implemented in OpenGL using extensions such as shadow buffering. [42]

3.5 Graphics Hardware CSG Rendering

The CSG rendering problem is a generalisation of the classical visible surface problem [92]: given a collection of geometry related by volumetric union, intersection and difference, which parts of which shapes are visible for a particular viewing situation? CSG rendering can be regarded as two sub-problems: clipping and visible surface determination. Clipping in this context is the process of identifying surfaces satisfying the constraints of the CSG tree. Visible surface determination is the process of identifying the visible surfaces with respect to a particular viewing situation.

Rendering an image of a CSG tree can be approached in different ways. *Object-space* approaches, such as boundary evaluation [74, 75, 17, 45, 40, 80] can be used to convert from a CSG representation to a boundary representation (b-rep), which is typically polygonal, that can be passed directly to the OpenGL rendering pipeline. With this approach the clipping problem is solved view-independently, and the visible surface problem solved per frame using standard z-buffering. *Image-space* approaches solve clipping and visible surface problems on a per-pixel basis — determining the visible surface for each pixel according to the viewing situation, geometric shapes involved and the CSG tree structure. Image-space approaches include ray-casting [47], scanline algorithms [16, 68, 50, 48, 49] and hardware-based z-buffer algorithms [79, 28, 37, 97, 73]. One attraction of image-space approaches is the utilisation of parallel graphics hardware rather than serial CPU boundary evaluation. Image-space CSG rendering is especially suited to interactive and animation applications that would otherwise require CPU-intensive object-space boundary evaluation for each and every frame.

OpenGL does not directly support CSG rendering. However, CSG rendering algorithms can be devised which use multi-pass techniques and features of the OpenGL API such as depth testing, stencil testing, buffer copying and back face culling. Image-space CSG rendering algorithms are the focus of this research.

The next two sections describe previous techniques for image-space CSG rendering: the Goldfeather algorithm [37, 97] and the Trickle algorithm [28]. Our new SCS algorithm is introduced in Chapter 4.

3.5.1 Goldfeather CSG Rendering

The Goldfeather CSG rendering algorithm [36, 37] clips one primitive in the z-buffer at a time. A second z-buffer is used to determine the visible surface in the usual *z-less* test manner. The algorithm assumes that the CSG tree is converted to *sum-of-products* (Section 2.2.1) form. A CSG tree can be *normalised* [36, 37] into sum-of-products form: $P_1 \cup P_2 \cup \cdots \cup P_p$, where p is the number of products. Each product consists of objects related by only intersection and difference operations.

The Goldfeather CSG rendering algorithm is presented in Algorithm 3.5. Using the CSG tree in Figure 2.3, the operation of the Goldfeather algorithm is illustrated in Figure 3.6. Each row corresponds to a pass of the algorithm. To begin with, the current primitive (a) is drawn into the z-buffer with front or back face culling enabled (b), according to whether a difference or intersection is being applied. Then, the z-buffer surface is parity clipped against all other primitives in the product (c). Parity clipping was described previously in Section 3.4.4. Finally the clipped z-buffer is merged into the output z-buffer (d).

Concave primitives are handled by clipping each *layer* (Section 3.4.2) 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. 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.

Algorithm 3.5 Goldfeather CSG Rendering

```
Require: all objects are convex
Require: $z_{\text{output}}$ is output z-buffer
Require: $z_{\text{tmp}}$ is temporary z-buffer

  $z_{\text{output}} \leftarrow z_{\text{far}}$
  for all products $P$ do
    for all objects $O \in P$ do
      if $O$ subtracted then
        $z_{\text{tmp}} \leftarrow O_{\text{back}}$
      if $O$ intersected then
        $z_{\text{tmp}} \leftarrow O_{\text{front}}$
      for all other objects $C \in P - O$ do
        clip $z_{\text{tmp}}$ against $C$ [ Algorithm 3.4 ]
      for all pixels do
        if $z_{\text{tmp}} < z_{\text{output}}$ then
          $z_{\text{output}} \leftarrow z_{\text{tmp}}$
```

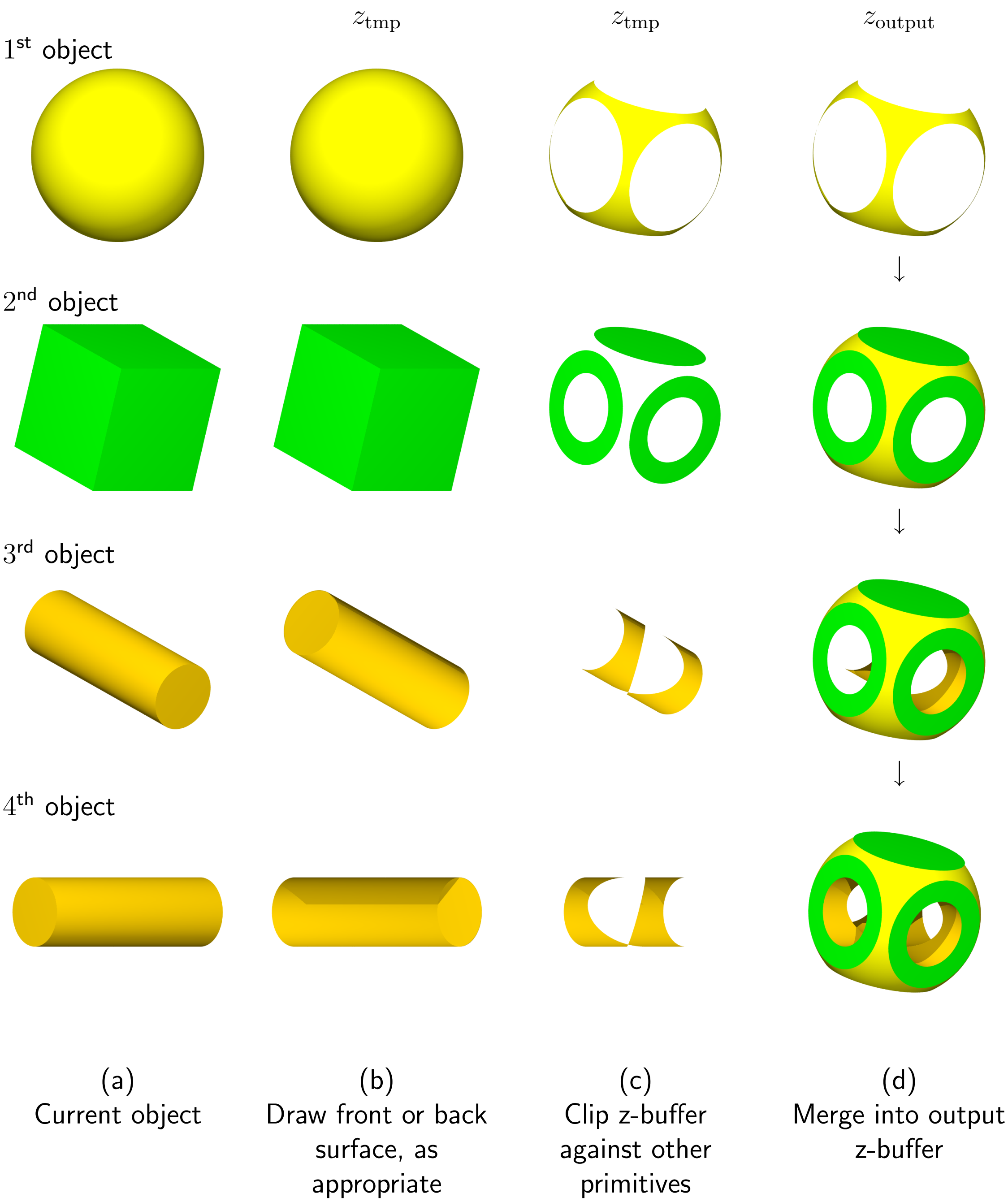


Figure 3.6: Goldfeather CSG Rendering Algorithm

The Goldfeather algorithm has been implemented using OpenGL [87, 97]. Two z-buffers are required by the algorithm, one for clipping each primitive and another for accumulating the merged result. The additional z-buffer can be simulated by copying between the z-buffer and system memory, or by subdividing the viewport. 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 [97].

3.5.2 Layered Goldfeather Algorithm

Graphics hardware based implementations of the Goldfeather algorithm utilise buffer copying to emulate the necessary additional z-buffer. Using this approach, z-buffer copying has been observed to be a significant performance bottleneck [87, 97].

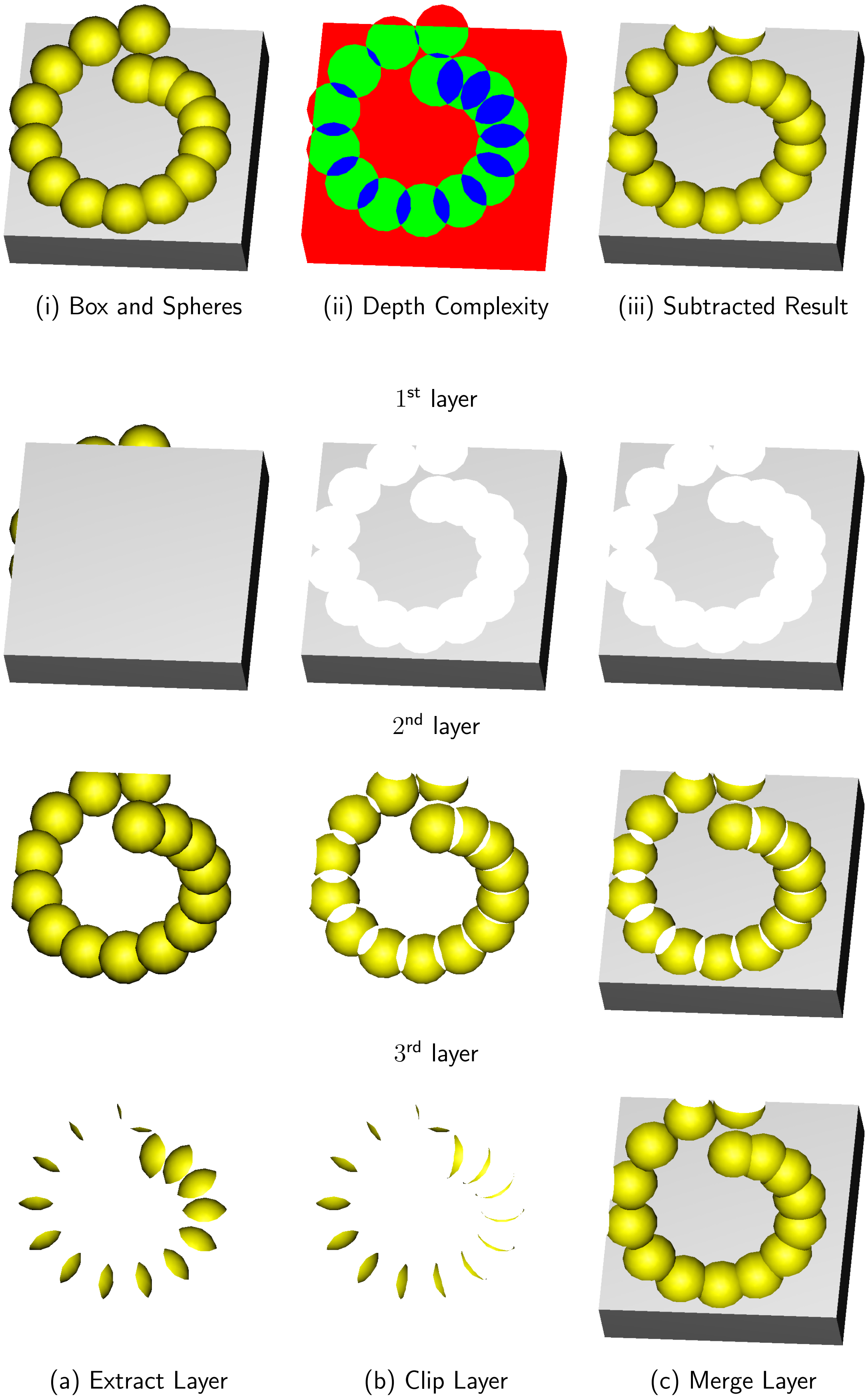
The Layered Goldfeather algorithm [87] aims to reduce buffer copying by clipping layers, rather than primitives. Denoting the depth complexity k, the algorithm becomes $O(kn)$, rather than $O(n^2)$. The z-buffer is copied once per layer, rather than once per primitive. When the depth complexity k is less than n, major speedups are realised by the reduction in z-buffer copying.

Figure 3.7 illustrates the operation of the algorithm (Algorithm 3.6), clipping on a layer-by-layer basis, rather than primitive-by-primitive: (i) a set of yellow spheres and a grey box; (ii) the depth complexity varies between zero and three; (iii) the yellow spheres subtracted from the grey box; (a) each layer is drawn into the z-buffer; (b) each layer is clipped against the spheres and box; (c) each layer is then merged into z_{output}.

Our implementation of this variant of the Goldfeather algorithm is given in Appendix B.2.1.

Algorithm 3.6 Layered Goldfeather CSG Rendering

```
Require: Objects are convex or concave
Require: $z_{\text{output}}$ is output z-buffer
Require: $z_{\text{tmp}}$ is temporary z-buffer

  $z_{\text{output}} \leftarrow z_{\text{far}}$

  for all products $P$ do
    for all layers $L_i \in L_1, L_2, \cdots, L_k$ of $P$ do
      $z_{\text{tmp}} \leftarrow L_i$
      for all objects $C \in P$ do
        clip $z_{\text{tmp}}$ against $C$ [Algorithm 3.4 ]
      for all pixels do
        if $z_{\text{tmp}} < z_{\text{output}}$ then
          $z_{\text{output}} \leftarrow z_{\text{tmp}}$
```



(i) Box and Spheres

(ii) Depth Complexity

(iii) Subtracted Result

1st layer

2nd layer

3rd layer

(a) Extract Layer

(b) Clip Layer

(c) Merge Layer

Figure 3.7: Layered Goldfeather CSG rendering algorithm

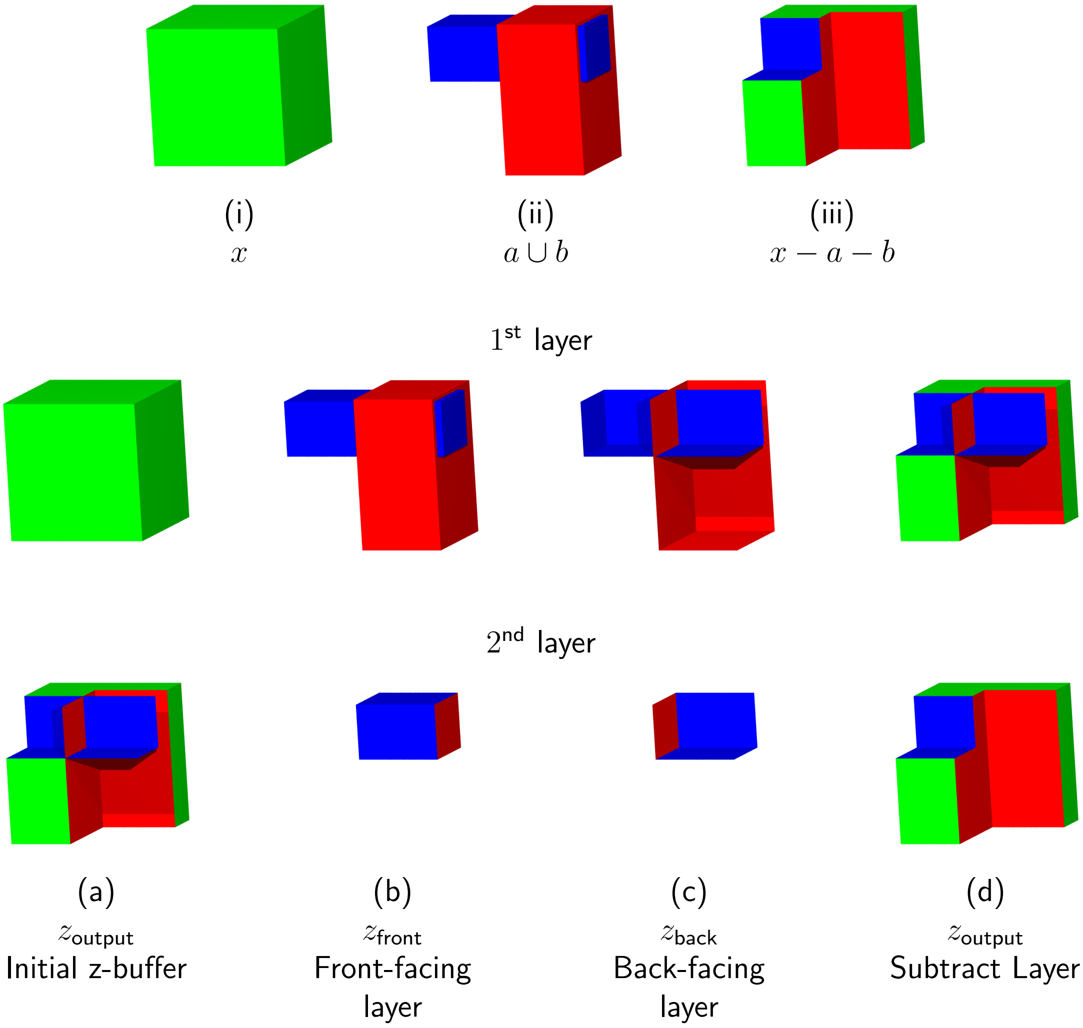
3.5.3 Improved Layered Goldfeather Algorithm

A further improvement to the Layered Goldfeather CSG rendering algorithm (Algorithm 3.7) is storing the result of layer clipping in the stencil buffer, rather than the depth buffer [29]. This is a performance advantage since stencil information is more compact than depth buffer information, and can be copied more efficiently. It is also has the advantage that integer stencil data is not subject to the implicit conversions that depth data is.

Our implementation of this variant of the Goldfeather algorithm is also given in Appendix B.2.2. In our implementation the depth buffer is used for merging each product, although in principle the stencil buffer could be used instead.

Algorithm 3.7 Improved Layered Goldfeather CSG Rendering

```
Require: Objects are convex or concave
Require: $z_{\text{output}}$ is output z-buffer
Require: $z_{\text{tmp}}$ is temporary z-buffer
Require: $stencil_i$ is the stencil bitmask for the $i^{\text{th}}$ layer, where $i \in [1, k]$

  $z_{\text{output}} \leftarrow z_{\text{far}}$

  for all products $P$ do

    for all layers $L_i \in L_1, L_2, \cdots, L_k$ do
      $z_{\text{tmp}} \leftarrow L_i$
      for all objects $C \in P$ do
        clip $z_{\text{tmp}}$ against $C$
      for all pixels do
        if $z_{\text{tmp}} < z_{\text{far}}$ then
          $stencil_i \leftarrow true$
        else
          $stencil_i \leftarrow false$

    for all layers $L_i \in L_1, L_2, \cdots, L_k$ do
      for all fragments $F \in L_i$ do
        if $z_F < z_{\text{output}}$ and $stencil_i = true$ then
          $z_{\text{output}} \leftarrow z_F$
```



(i) x

(ii) $a \cup b$

(iii) $x - a - b$

1st layer

2nd layer

(a) z_{output} Initial z-buffer

(b) z_{front} Front-facing layer

(c) z_{back} Back-facing layer

(d) z_{output} Subtract Layer

Figure 3.8: The Trickle CSG Rendering Algorithm

3.5.4 Trickle Algorithm

The Trickle CSG rendering algorithm [28] subtracts volumetric layers from front to back with respect to the viewing direction. Four depth buffers are used: two auxiliary z-buffers for peeling the front and back layers, a temporary scratch-space z-buffer and a z-buffer storing the cumulative result. The Trickle algorithm is designed to handle CSG products. CSG products contain intersection and subtraction operations, but no unions. CSG products are in the form: $x \cap o_1 \cap \cdots \cap o_p - o_{p+1} - \cdots - o_n$. The algorithm operates on a subtractive basis with intersection handled by subtracting inverted shapes.

Figure 3.8(a–d) illustrates the application of the Trickle algorithm to a CSG tree of three boxes ($x - a - b$) as illustrated in Figure 3.8(i–iii). Initially the front of x is drawn into the z-buffer (a). Then, the closest front-facing (b) and closest back-facing (c) surfaces of subtracted objects are determined for each pixel. This forms a volumetric layer, which

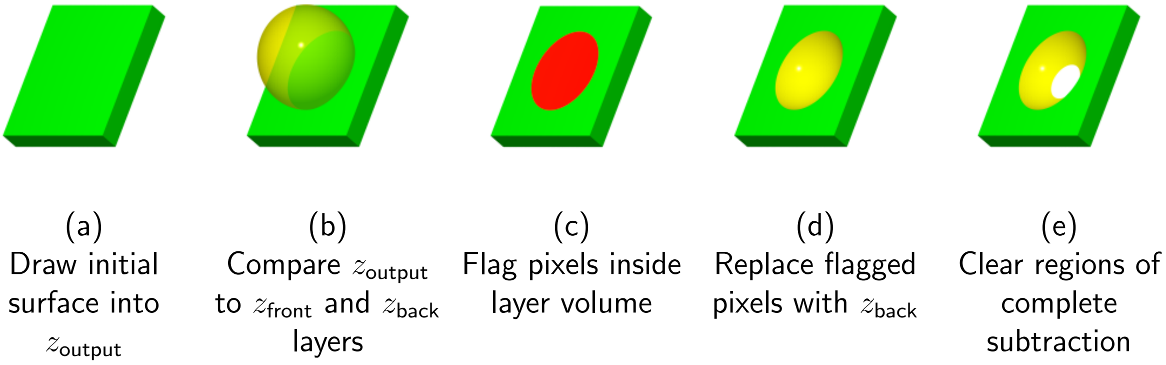

(a)
Draw initial surface into z_{output}
(b)
Compare z_{output} to z_{front} and z_{back} layers
(c)
Flag pixels inside layer volume
(d)
Replace flagged pixels with z_{back}
(e)
Clear regions of complete subtraction

Figure 3.9: Depth Buffer Convex Subtraction

is subtracted from the output z-buffer (d). Subtraction replaces the z-buffer with z_{back}, the back-facing surface of the layer, for each pixel where $z_{\text{front}} < z_{\text{output}} < z_{\text{back}}$. The next layer is then 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 front to back. Layers are view-dependent, since changing the viewing direction affects the relative distances of surfaces to the viewer.

The subtraction of an individual convex object or volumetric layer is illustrated in Figure 3.9. The Trickle algorithm repeats steps (b–d) for each layer of subtracted volume.

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

The complete Trickle algorithm is presented in Algorithm 3.8. The near and far clipping planes of the viewing system are denoted as z_{near} and z_{far}. Layer peeling of front and back surfaces operates as previously described in Section 3.4.3.

Intersections are handled by subtracting inverted primitives. The inverse of a convex primitive S spanning $[S_{\text{front}}, S_{\text{back}}]$ is $[z_{\text{near}}, S_{\text{front}}]$ and $[S_{\text{back}}, z_{\text{far}}]$. Implementing this inversion involves swapping the front and back-facing surfaces, and manipulating the stencil buffer. 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}.

Algorithm 3.8 Trickle CSG Rendering

```
Require: $X$ is convex object
Require: $z_{\text{output}}$ is output z-buffer
Require: $z_{\text{front}}$ is front layer z-buffer
Require: $z_{\text{back}}$ is back layer z-buffer
  [Draw initial surface — as illustrated in Figure 3.9(a)]
  $z_{\text{output}} \leftarrow z_{\text{far}}$
  draw $X_{\text{front}}$ into $z_{\text{output}}$
  [Layer peeling loop]
  $z_{\text{front}} \leftarrow z_{\text{near}}$
  $z_{\text{back}} \leftarrow z_{\text{near}}$
  for all layers do
    $z_{\text{front}} \leftarrow peelLayer(z_{\text{front}})$ [Algorithm 3.3 ]
    $z_{\text{back}} \leftarrow peelLayer(z_{\text{back}})$
    [Subtract volumetric layer — as illustrated in Figure 3.9(b–d)]
    for all pixels do
      if $z_{\text{front}} < z_{\text{output}} < z_{\text{back}}$ then
        $z_{\text{output}} \leftarrow z_{\text{back}}$
  [Complete subtraction — as illustrated in Figure 3.9(e)]
  for all pixels do
    if $z_{\text{output}} > X_{\text{back}}$ then
      $z_{\text{output}} \leftarrow z_{\text{far}}$
```

3.6 Discussion

This chapter has introduced aspects of the OpenGL Graphics API and discussed previously reported CSG rendering algorithms. These form the foundation for our *Sequenced Convex Subtraction* (SCS) CSG rendering algorithm presented in the following chapters.

The Goldfeather or Trickle algorithms were devised independently to the OpenGL graphics architecture, and each pose performance issues in the context of OpenGL 1 graphics hardware. The Goldfeather algorithm relies heavily on depth buffer copying in OpenGL, leading to revised versions of the algorithm [87, 29] making alternative trade-offs in order to increase overall performance. The Trickle algorithm relies on functionality not commonly found in graphics hardware. Limitations and performance characteristics of graphics hardware is intricately related to the appropriate approach to image-space CSG rendering for a particular application.

The overall progression in image-space CSG rendering techniques is the use of increasingly sophisticated algorithms. The increasing sophistication of graphics hardware may lead to even further complexity in approaches to CSG rendering. Alternatively, certain innovations may resolve certain bottlenecks and limitations, facilitating simpler approaches to CSG. In this work we have tended towards relatively sophisticated use of well established OpenGL functionality with the aim of maximising performance. With graphics hardware rapidly evolving the ideal long-term solution to interactive CSG rendering is not yet clear.

Chapter 4

Sequenced Convex Subtraction

Engineering is the professional art of applying science to the optimum conversion of natural resources to the benefit of man.
— Ralph J. Smith

4.1 Introduction

This chapter introduces a new algorithm for hardware based image-space CSG rendering that we call *Sequenced Convex Subtraction* (SCS). It is based on the subtraction of convex objects from the z-buffer and linear time intersection of convex objects. The presentation of SCS is based on our previous publications: *A Z-Buffer CSG Rendering Algorithm for Convex Objects* [88] and *Linear-time CSG Rendering of Intersected Convex Objects* [89]. We describe the methodology and implementation of the SCS algorithm in the context of previous CSG rendering algorithms and hardware considerations. Techniques including image-space convex intersection, convex subtraction, depth complexity sampling and subtraction sequences are combined to form the SCS multi-pass algorithm for CSG rendering.

Extensions to the SCS algorithm are discussed in subsequent chapters. Chapter 5 focuses on the sequences used by the SCS algorithm for CSG subtraction. Chapter 6 presents an extension to SCS making use of object-space intersection information to improve performance. Chapter 7 examines performance aspects of the SCS algorithm.

4.1.1 Previous Algorithms

As described in Chapter 3, previous hardware based CSG rendering algorithms include the Trickle [28] algorithm and the Goldfeather [36, 37, 97] algorithm. The strategy of the Trickle algorithm is to sort subtracted objects into volumetric layers from front to back and to subtract each layer in turn. As the algorithm "trickles" from front to back, a subtracted layer may reveal a subsequent layer further away from the viewer. Once the current layer is further than the z-buffer, no subsequent layers that are further away can be visible to viewer since they are occluded by the z-buffer surface.

The SCS algorithm operates in a similar manner to the Trickle algorithm except that convex objects are subtracted, rather than sorted image-space layers composed of multiple convex or concave objects. Instead of sorting convex objects from back to front, a subtraction sequence is determined which ensures that the necessary sequence of subtractions is performed.

The strategy of the Goldfeather algorithm is to clip each object, or each layer of concave objects in the z-buffer. These clipped z-buffers are then merged into a combined result, and finally drawn to the colour buffer.

Like previous ones, the SCS algorithm makes use of CSG tree normalisation, the process of converting an arbitrary CSG tree into sum-of-products form. In this way, intersection and subtraction of objects in each product are resolved by the SCS algorithm, and products are merged in z-buffer form in a manner similar to the Goldfeather algorithm.

The SCS algorithm is restricted to CSG trees consisting of only convex objects. This facilitates the use of methods less complicated and more efficient than would be otherwise possible. However, it requires that concave objects are decomposed into convex components by the application, rather then being handled directly by the rendering algorithm. The *Extended Convex Differences Tree* (ECDT) CSG approach[72] also requires conversion to a convex representation. The Trickle and Goldfeather approaches handle concave objects directly, as part of the rendering algorithm. One of the broader considerations in this work is determining the best arrangement of CPU and graphics hardware computation for a particular task.

4.1.2 Hardware Considerations

The Trickle algorithm requires the use of multiple depth buffers and multiple simultaneous z-tests. The front and back surfaces of the current volumetric layer are represented by separate depth buffer layers. An additional depth buffer and a multiple z-test is used in the process of determining the front or back surface of the next layer. A fourth depth buffer is used to represent the final result of the subtracted layers. The OpenGL 1 standard does not include multiple depth buffers or depth tests. Emulation of these features by buffer copying and extra passes impacts substantially on the performance of the Trickle algorithm on current graphics hardware.

The Goldfeather algorithm requires two depth buffers, one for clipping objects and another for storing the accumulated result. The overhead involved in simulating two depth buffers can become a bottleneck for OpenGL implementations of the algorithm [97, 87].

The SCS CSG rendering algorithm aims to achieve improved performance by better utilising the features and constraints of commodity graphics hardware. The algorithm was designed with the following assumptions:

- Rasterisation is faster than buffer copying.

- Rasterisation is better supported, more robust and optimised at the driver and hardware level than is buffer copying.

- Rasterisation performance is likely to improve more rapidly than buffer copying bandwidth.

- Image-space sorting requires $O(n^2)$ or $O(kn)$ passes.

- Rendering of individual CSG products has useful applications.

The evolution of graphics hardware has seen consistent emphasis on rasterisation performance in terms of triangles per second, texturing, multi-texturing and programmable per-pixel shading. While buffer-copying bandwidth has also improved over time, it has not been considered as important for mainstream applications of graphics hardware such as games. It appears that the economics of dedicating additional rasterisation features and performance have been more attractive than increasing the performance of copying in and out of frame buffer memory.

The SCS algorithm is particularly efficient for rendering individual CSG products since no buffer copying is required, in contrast to the Trickle or Goldfeather algorithms. This work was originally motivated by the need to visualise a tool grinding manufacturing process that typically involves many subtractions from an initial convex shape. We consider the SCS approach particularly well suited to this type of application.

Over the longer term graphics hardware functionality is expected to increase and may become better suited to the Trickle or Goldfeather approaches to CSG rendering. While the SCS algorithm may also benefit by being extended or enhanced to utilise new hardware capability, it is hard to predict into the future which algorithmic approach will be best suited to the hardware of the time.

4.1.3 Presentation of the SCS Algorithm

The following sections describe individual stages of the *Sequenced Convex Subtraction* (SCS) image-space CSG rendering algorithm. Section 4.2 introduces the frame buffer operations utilised by the SCS CSG rendering algorithm. These include intersection of convex objects, subtraction of convex objects, depth buffer clipping, product merging and a final pass for lighting and shading. Section 4.3 introduces the sequences of subtraction that are used by the algorithm as the alternative to depth sorting. These sequences can be either view-dependent or view-independent. Section 4.4 combines the frame buffer and subtraction sequence concepts into the overall SCS CSG rendering algorithm.

4.2 Frame Buffer Operations

4.2.1 Convex Intersection

CSG products contain at least one intersected primitive, from which other primitives are subtracted. The first step of the SCS algorithm is to draw the result of intersected primitives into the z-buffer. Subsequent stages of the algorithm deal with subtraction and union operations.

The 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 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 not possibly be inside the intersection. Consequently, if the depth-complexity of a pixel is less than n then it can not possibly be inside all n objects.

The z-buffer image of a convex CSG intersection can be determined in linear time with respect to the number of intersected primitives. To begin with, the z-buffer is initialised to z_{near}, and the stencil buffer reset to zero. The furthest front facing surface is then drawn

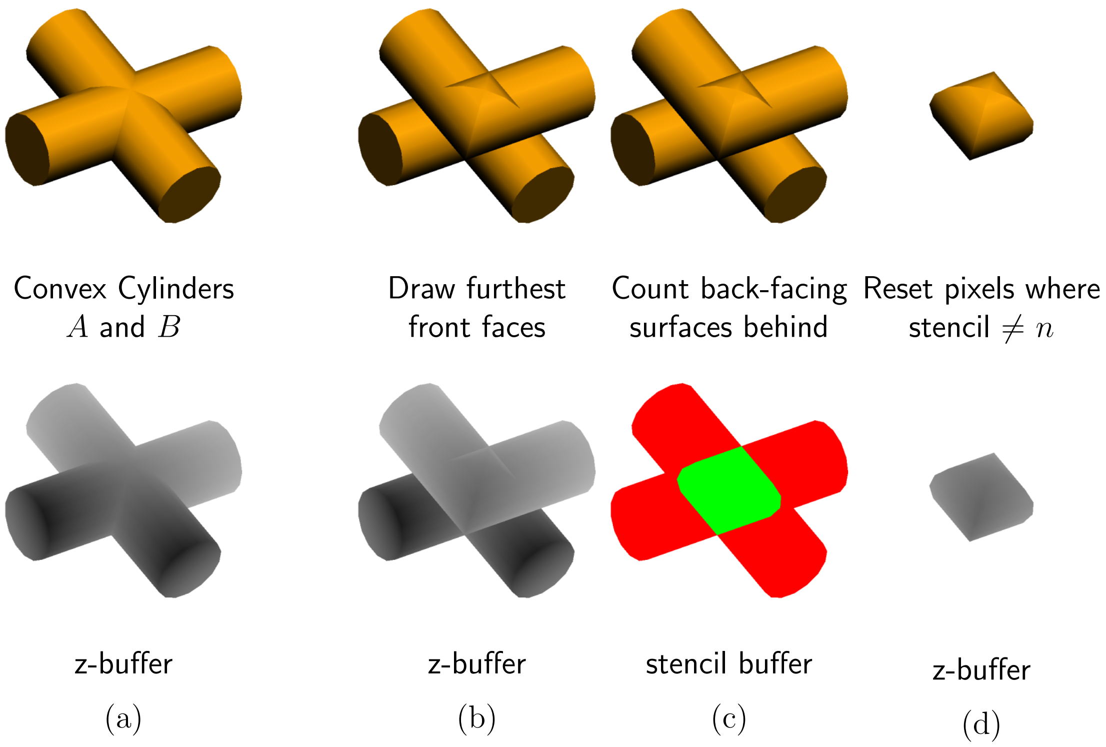


Figure 4.1: Depth Buffer Convex Intersection

into the z-buffer, by rasterising with a z-greater test. The stencil buffer is then used to count the number of back-facing primitives behind the furthest front facing surface in the z-buffer. Finally, all pixels that do not have n back-facing surfaces behind the z-buffer are reset to z_{far}.

Figure 4.1 illustrates the intersection of two cylinders. The z-buffer illustrated in (b) is clipped according to the stencil in (c) to produce the depth buffer of the intersected cylinders in (d). In the z-buffer diagrams grey level indicates depth with z_{near} black and z_{far} white. White, red and green denote values of zero, one and two in the stencil-buffer.

The SCS Convex Intersection algorithm is described in Algorithm 4.1. The algorithm is linear time since each of the front and back facing surfaces of each primitive is drawn once. In the context of the SCS rendering algorithm it is not necessary to draw the colour image of the CSG intersection, although the algorithm can be modified to do so.

Our C++ OpenGL implementation appears in Appendix B.3.1. 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 most common case, 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.

Algorithm 4.1 SCS Convex Intersection Rendering

```
Require: Objects are convex
Require: $n$ is the number of intersected objects
  for all pixels do
    $z \leftarrow z_{\text{near}}$
    $stencil \leftarrow 0$
  for all objects $O$ do
    for all front fragments $F_{\text{front}}$ of $O$ do
      if $F_{\text{front}} > z$ then
        $z \leftarrow F_{\text{front}}$
  for all objects $O$ do
    for all back fragments $F_{\text{back}}$ of $O$ do
      if $F_{\text{back}} > z$ then
        $stencil \leftarrow stencil + 1$
  for all pixels do
    if $stencil \neq n$ then
      $z \leftarrow z_{\text{far}}$
```

4.2.2 Convex Subtraction

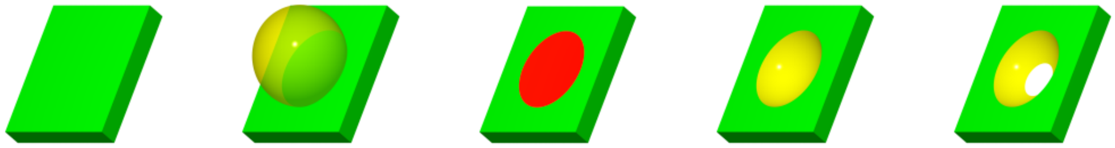
Subtracted primitives in a CSG product are handled by sequenced subtraction from the z-buffer. Each subtraction involves comparing the front and back facing surfaces to the z-buffer. The z-buffer is updated for each pixel volumetrically inside the subtracted primitive. The algorithm is given in Algorithm 4.2.

Algorithm 4.2 SCS Convex Subtraction Rendering

```
Require: Object $O$ is convex
Require: z-buffer contains front surface of intersected primitives
  for all front fragments $F_{\text{front}}$ of $O$ do
    if $F_{\text{front}} < z$ then
      $stencil \leftarrow 1$
    else
      $stencil \leftarrow 0$
  for all back fragments $F_{\text{back}}$ of $O$ do
    if $F_{\text{back}} > z$ and $stencil = 1$ then
      $z \leftarrow F_{\text{back}}$
```

Figure 4.2(b–d) illustrate the subtraction of a sphere from a rectangular block in the z-buffer. The OpenGL stencil buffer is used to flag pixels that are potentially inside the subtracted sphere. In a subsequent pass the z-buffer at these pixels is replaced with the back-facing surface of the subtracted sphere, wherever the back surface of the sphere is further than the depth buffer.

The z-buffer can store only one surface or layer at a time. Occluded cavities can not be represented. For this reason the SCS algorithm subtracts each primitive more than once to ensure that cavities revealed by other subtractions are correctly rendered. This sequenced subtraction approach is further discussed in Section 4.3.



(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 4.2: Depth Buffer Convex Subtraction

Our C++ OpenGL implementation appears in Appendix B.3.2. Steps (b–d) of Figure 4.2 are performed for each object in the subtraction sequence. Step (e) of Figure 4.2 is described in the following section — clipping the result of subtraction against the back faces of intersected objects, which we term z-buffer clipping.

4.2.3 Z-Buffer Clipping

As illustrated in Figure 4.3(b) the SCS algorithm first puts the front surface of the intersected objects into the z-buffer. In the second stage (c) subtracted objects are processed. In a third stage (d) the z-buffer is clipped against the back-facing surfaces of the intersected objects — identifying pixels of "complete subtraction".

In the z-buffer clipping step, pixels with z values further than any intersected back-facing surface are reset to z_{far}. This ensures that all remaining pixels are within the volume of all the intersected objects in the CSG product. The SCS z-buffer clipping algorithm is presented in Algorithm 4.3. The back-facing surface of each intersected object is rasterised and a stencil flag set for fragments closer than the current depth buffer.

So far, rendering a CSG product involves rasterising all of the front-facing surfaces of intersected objects, performing a sequence of subtractions, then rasterising all of the back-facing surfaces of intersected objects. The front and back surfaces of intersected objects are rasterised once, while subtracted objects are rasterised up to n times.

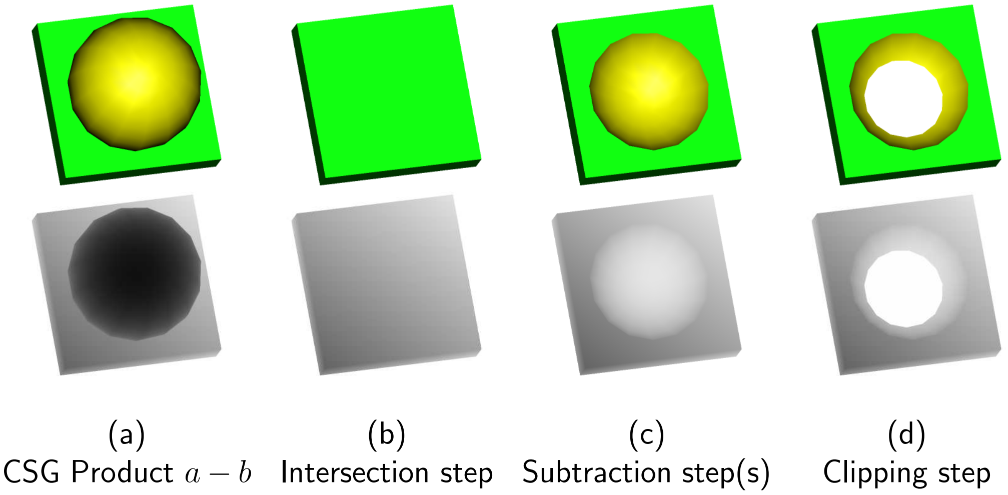


Figure 4.3: Z-Buffer Clipping Step of the SCS Algorithm

Algorithm 4.3 SCS Z-Buffer Clipping

```
Require: Objects are convex
Require: Stencil buffer is zero
  for all intersected objects $I$ do
    if $I_{\text{back}} < z$ then
      $stencil \leftarrow 1$
  for all pixels do
    if $stencil = 1$ then
      $z \leftarrow z_{\text{far}}$
    $stencil \leftarrow 0$
```

4.2.4 Z-Buffer Merging

Rendering of a CSG product is performed using the single z-buffer available in OpenGL graphics hardware. Rendering the union of multiple CSG products requires an additional z-buffer for storing the merged result of individual products. The merging algorithm is presented in Algorithm 4.4.

The OpenGL API provides two ways of simulating an additional z-buffer. The frame buffer can be copied into system memory, then later re-rasterised and depth-tested to merge it with the result of the next product. An alternative is to subdivide the viewport using half for the current product and then rasterising into the other half with depth-testing.

Either approach can be problematic, depending on the hardware platform. Copying between the frame buffer and system memory can be slow due to the limited bandwidth of the PCI or AGP bus. A round-trip from the z-buffer into system memory and back again can also result in truncation of precision — the internal graphics hardware z-buffer representation is not necessarily known to the application. The OpenGL implementation itself may also be applying conversions to z-buffer data as it is copied.

Copying z-buffer data within the frame buffer can also be slow, and can truncate precision. In our implementation we found this aspect of graphics hardware one of the most troublesome. We are optimistic that these issues will be resolved as hardware and

Algorithm 4.4 SCS Z-Buffer Merging

```
Require: OpenGL z-buffer $z$
Require: Second z-buffer $z_{\text{merged}}$
  for all pixels do
    if $z < z_{\text{merged}}$ then
      $z_{\text{merged}} \leftarrow z$
```

drivers evolve and mature.

4.2.5 Z-Buffer Shading and Lighting

The SCS algorithm focuses primarily on forming the depth buffer image of a CSG tree. The final step is to convert this representation to a fully lit and shaded rendered image in the colour buffer. This final pass of the SCS algorithm redraws all geometry with a z-equal depth test and lighting enabled.

The Z-Buffer Shading and Lighting algorithm is presented in Algorithm 4.5, and an OpenGL implementation is given in Appendix B.1.4. For performance reasons, rasterisation is constrained to the front-facing surfaces of intersected objects and back-facing surfaces of subtracted objects. Several CSG object representations are illustrated in Figure 4.4, along with their shaded and lit result.

Algorithm 4.5 SCS Z-Buffer Shading and Lighting

```
Require: Enable updates to colour buffer
Require: colour is the OpenGL colour buffer
  for all intersected primitives $I$ do
    for all front facing fragments $F_{\text{front}}$ of $I$ do
      if $F_{\text{front}} = z$ then
        $colour \leftarrow F_{\text{front}}$
  for all subtracted primitives $S$ do
    for all back facing fragments $F_{\text{back}}$ of $S$ do
      if $F_{\text{back}} = z$ then
        $colour \leftarrow F_{\text{back}}$
```

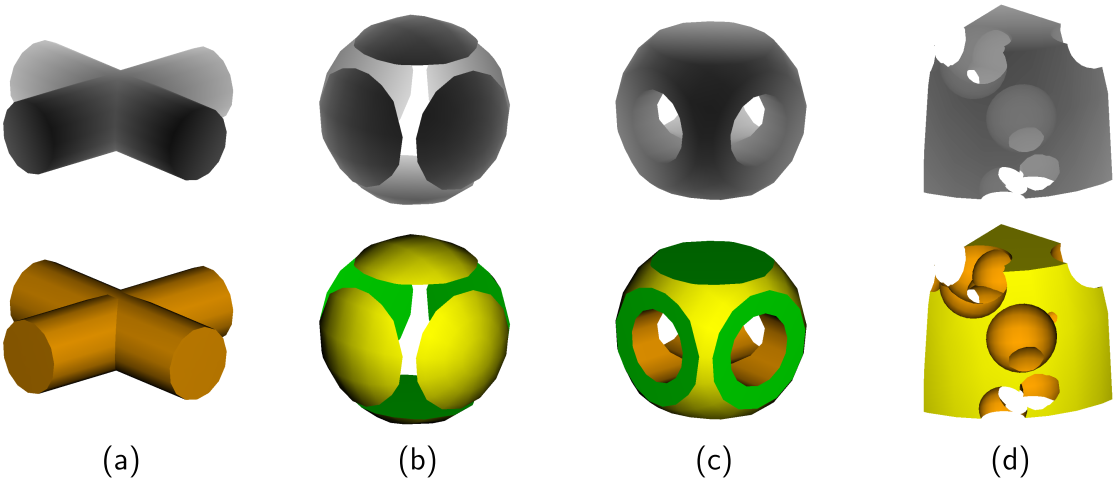


Figure 4.4: Z-Buffer Lighting and Shading

4.3 Subtraction Sequences

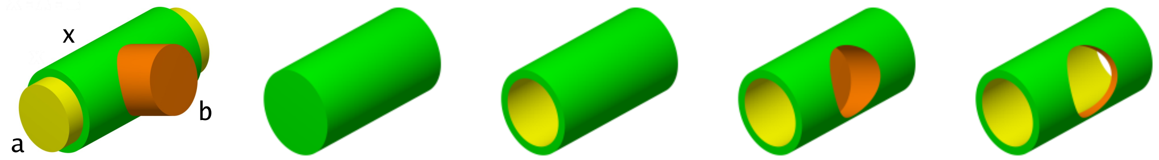
Subtraction sequences need to handle depth dependencies between subtracted primitives in the product. If a and b are overlapping subtracted primitives, subtracting a may reveal b or subtracting b may reveal a, depending on the viewing direction. The sequence needs to allow for both possibilities to ensure the correctly rendered result. The subtraction sequence aba incorporates both ab and ba without needing to know if a is in front or behind of b.

Figure 4.5 illustrates the subtraction of two cylinders in three subtraction steps. In Figure 4.5(c) a is subtracted, followed in Figure 4.5(d) by b. The subtraction of b reveals surfaces of a which can only be determined by subtracting a again in Figure 4.5(e). For any viewing direction or configuration of two subtracted primitives, the sequence $x - a - b - a$ ensures that both possible dependencies are correctly handled. The sequence $x - b - a - b$ also does.

Permutation Embedding Sequences [35] (PES) have the property that all $n!$ permutations of n objects are embedded. A sequence is embedded if it can be formed by deleting entries. For example, cab is embedded in the sequence $abcbabc$: $\star\star c\star ab\star$. Permutation embedding sequences of length $O(n^2)$ can be easily obtained by methods[88] described in the following sections. Sequences of shorter and shortest length are the focus of Chapter 5.

4.3.1 View Independent Sequences

This section describes a simple method for constructing a *view independent* permutation embedding sequence, given n subtracted objects in a scene. The sequence is said to be view independent because it may be used from any viewing direction. The next section examines *view dependent* permutation embedding sequences that may be shorter but must be regenerated for each viewing direction.



(a) CSG Tree

(b) x

(c) $x - a$

(d) $x - a - b$

(e) $x - a - b - a$

Figure 4.5: Subtraction Sequence for Two Objects

Sequence encoding 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:

$n = 2$, $s_1 = ab$

$s_1 \cdot s_1 \rightarrow ab\ ab \rightarrow abab$

$n = 3$, $s_1 = abc$

$s_1 \cdot s_1 \cdot s_1 \rightarrow abc\ abc\ abc \rightarrow abcabcabc$

$n = 4$, $s_1 = abcd$

$s_1 \cdot s_1 \cdot s_1 \cdot s_1 \rightarrow abcd\ abcd\ abcd\ abcd \rightarrow abcdabcdabcdabcd$

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. The length of these sequences is $n^2 - n + 1$.

For example:

$n = 2$, $s_1 = ab$, $s_2 = ba$

$s_1 \cdot s_2 \rightarrow ab\ ba \rightarrow aba$

$n = 3$, $s_1 = abc$, $s_2 = cba$

$s_1 \cdot s_2 \cdot s_1 \rightarrow abc\ cba\ abc \rightarrow abcbabc$

$n = 4$, $s_1 = abcd$, $s_2 = dcba$

$s_1 \cdot s_2 \cdot s_1 \cdot s_2 \rightarrow abcd\ dcba\ abcd\ dcba \rightarrow abcdcbabcdcba$

While this approach to permutation embedding sequence construction is straight forward, shorter sequences can be constructed using other methods [14, 35, 83] described in Chapter 5. The length of sequences using these other methods is $n^2 - 2n + 4$.

4.3.2 View Dependent Sequences

The depth complexity of a scene is the maximum number of objects covering an individual pixel, as previously discussed in Section 3.4.1. By determining the depth complexity of a scene from a particular viewing direction, a shorter view dependent subtraction sequence can be used rather than a view independent sequence.

Denoting k as the maximum number of subtracted objects overlapping any pixel, subtraction sequences of $O(kn)$ length can be used, rather than $O(n^2)$. In the best case $k = 1$, and for convex objects in the worst case $k = n$. Therefore, the advantage of view

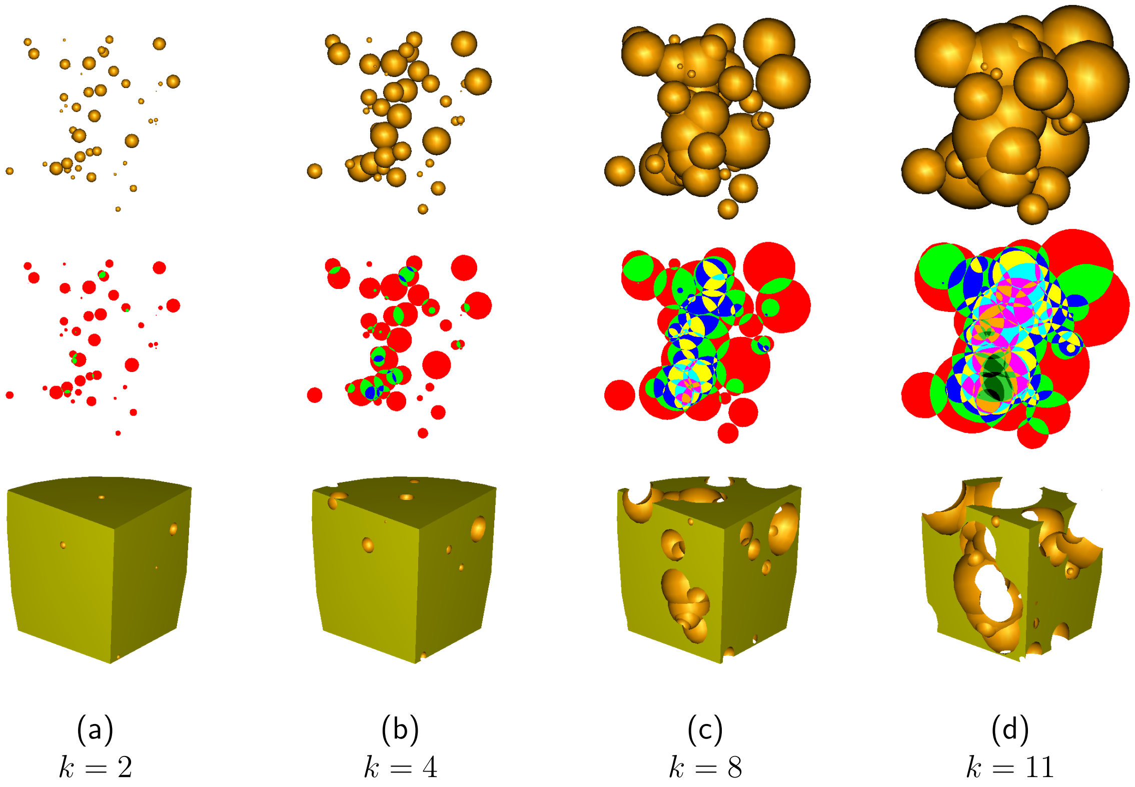


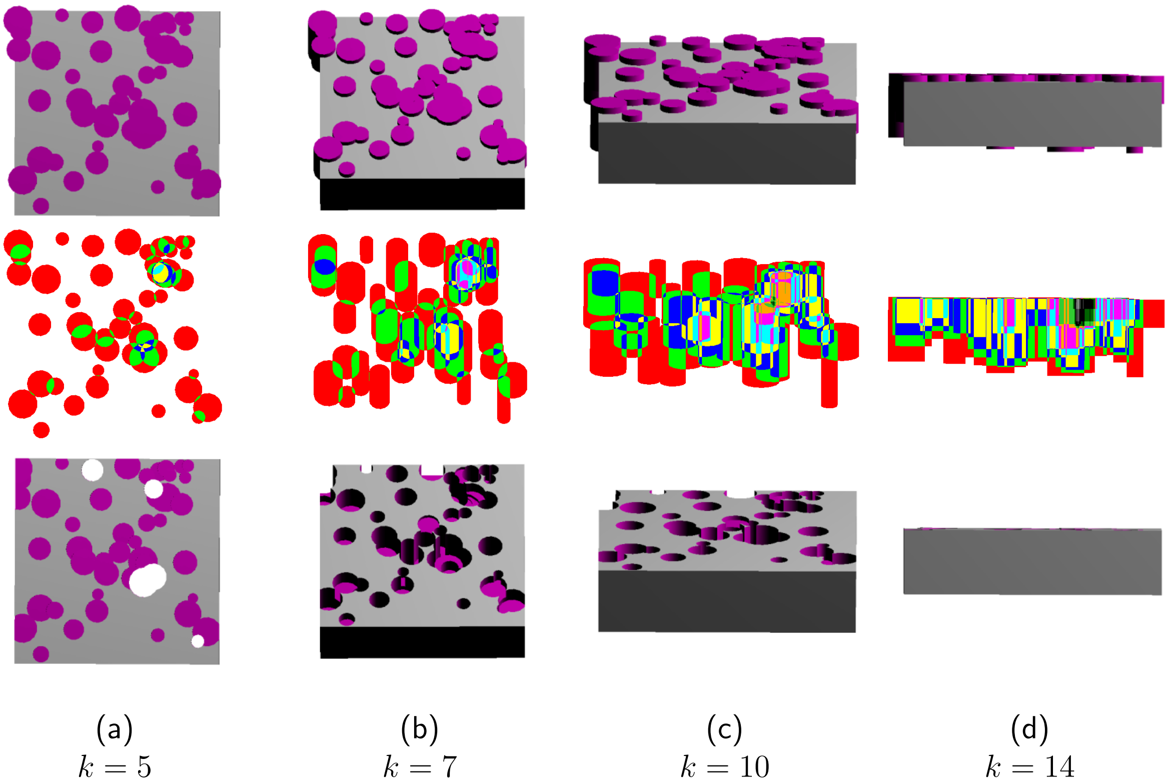
Figure 4.6: Depth Complexity of CSG Cheese Model with 50 Holes

dependent subtraction sequences depends on the particular arrangement of objects and the viewing direction. Whilst view dependent subtraction sequences may be shorter and are never longer than a view independent sequence, there may be additional overhead involved in sampling the depth complexity in order to determine them.

Figure 4.6 illustrates a procedural swiss cheese CSG model with 50 holes of varying size. As the holes become larger from (b) to (d), overlap between holes and the depth complexity increases, resulting in longer view dependent subtraction sequences.

Figure 4.7 illustrates the depth complexity of a simulated 3-axis milling model from different viewing directions. Viewed from above in (a) the depth complexity is 5, while viewed from the side in (d) the depth complexity is 14.

The following method for constructing view dependent subtraction sequences extends the view independent method from the previous section. As previously, two permutations s_1 and s_2 are used with s_2 being the reversal of s_1. The sequence is formed by k alternating copies of s_1 and s_2 with repeated entries between adjacent copies collapsed. All k-permutations are embedded — the permutations of k selections from n objects.



(a) $k = 5$

(b) $k = 7$

(c) $k = 10$

(d) $k = 14$

Figure 4.7: Depth Complexity of Milling Model from Different Viewing Directions

For example:

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

$s_1 \cdot s_2 \longrightarrow abc\ cba \longrightarrow abcba$

$n = 4,\ k = 2,\ s_1 = abcd,\ s_2 = dcba$

$s_1 \cdot s_2 \longrightarrow abcd\ dcba \longrightarrow abcdcba$

$n = 4,\ k = 3,\ s_1 = abcd,\ s_2 = dcba$

$s_1 \cdot s_2 \cdot s_1 \longrightarrow abcd\ dcba\ abcd \longrightarrow abcdcbabcd$

Methods for determining view-dependent sequences of length $kn - 2k + 4$ are examined in Chapter 5. In Chapter 6 it is shown that even shorter subtraction sequences can be constructed using information about the spatial overlap of subtracted objects.

4.4 SCS Algorithm

4.4.1 Complete Algorithm

In previous sections the image-based algorithms for CSG intersection, subtraction and depth buffer clipping were described. Here these are combined into the overall *Sequenced Convex Subtraction* (SCS) CSG rendering algorithm which renders a CSG tree in sum-of-products form using the OpenGL graphics pipeline.

Each CSG product in the sum is processed in a separate pass. Initially, the intersected objects in the product are drawn into the OpenGL depth buffer, as described in Section 4.2.1. Then the subtracted objects in the product are subtracted sequentially as described in Section 4.2.2 using a sequence as discussed in Section 4.3. The depth buffer is then clipped against the back faces of the intersected objects, as described in Section 4.2.3. If necessary, the depth buffer is finally merged with the result of the previous CSG products as described in Section 4.2.4. Once the depth buffer of the CSG tree has been determined, the shaded and lit result is drawn into the colour buffer, as described in Section 4.2.5.

The complete algorithm is presented in Algorithm 4.6. The second depth buffer is only necessary if there is more than one CSG product to be processed.

Algorithm 4.6 SCS CSG Rendering

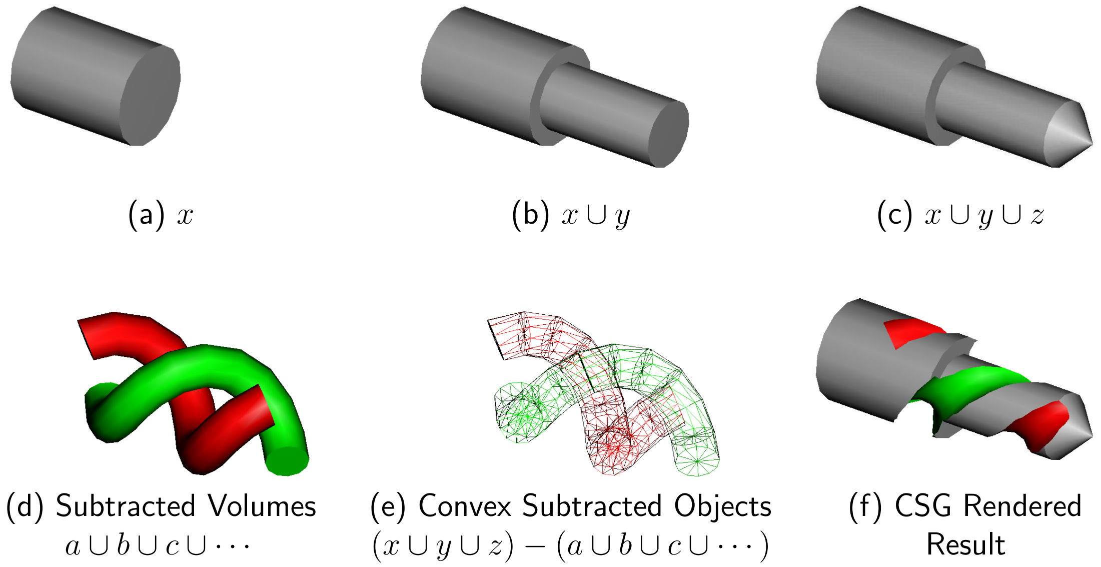
```
Require: Objects are convex
Require: CSG tree in sum-of-products form
Require: Colour buffer: $colour$
Require: Primary z-buffer: $z$
Require: Secondary z-buffer: $z_{\text{merged}}$

  [Clear the frame buffer]
  $colour \leftarrow$ background colour
  $z_{\text{merged}} \leftarrow z_{\text{far}}$

  [Loop over products]
  for all CSG products do
    do SCS convex intersection                    [Algorithm 4.1 ]
    determine subtraction sequence
    do SCS convex subtraction                     [Algorithm 4.2 ]
    do z-buffer clipping                          [Algorithm 4.3 ]
    merge $z$ into $z_{\text{merged}}$            [Algorithm 4.4 ]

  [Convert z-buffer to colour buffer — Algorithm 4.5]
  do depth buffer shading and lighting
```

4.4.2 Example

The overall strategy of the SCS CSG rendering algorithm is illustrated in Figure 4.8 and Figure 4.9. For illustrative purposes, a simple CSG machine tool is composed of two cylindrical sections, a conical tip and two helical swept volumes decomposed into convex segments. The helical volumes correspond to the volume that would be subtracted by the movement of a cutting tool over time.

In normalised form, the CSG model consists of three products, each composed of subtractions of convex objects from a convex segment of the initial shape. Each product is processed by the SCS algorithm in a separate pass, as illustrated in Figure 4.9. In each pass, (a) the z-buffer intersection is formed in the depth buffer, (b) a subtraction sequence is determined and subtractions are performed on the depth buffer, (c) the depth buffer is clipped against the back-faces of the intersecting objects, and (d) the product depth buffer is merged with the cumulative result of other product passes. Finally in (e), the combined depth buffer is lit and shaded by redrawing all the geometry with a `GL_EQUAL` depth test.

This strategy for assembling the final sum-of-products result resembles the Goldfeather algorithm as previously described in Section 3.5.1 and illustrated in Figure 3.6. While the Goldfeather algorithm clips and merges each shape in each product, the SCS algorithm only performs one merge per product. The performance implications of this are examined in Chapter 7.



(a) x

(b) $x \cup y$

(c) $x \cup y \cup z$

(d) Subtracted Volumes $a \cup b \cup c \cup \cdots$

(e) Convex Subtracted Objects $(x \cup y \cup z) - (a \cup b \cup c \cup \cdots)$

(f) CSG Rendered Result

Figure 4.8: CSG Machine Tool Example

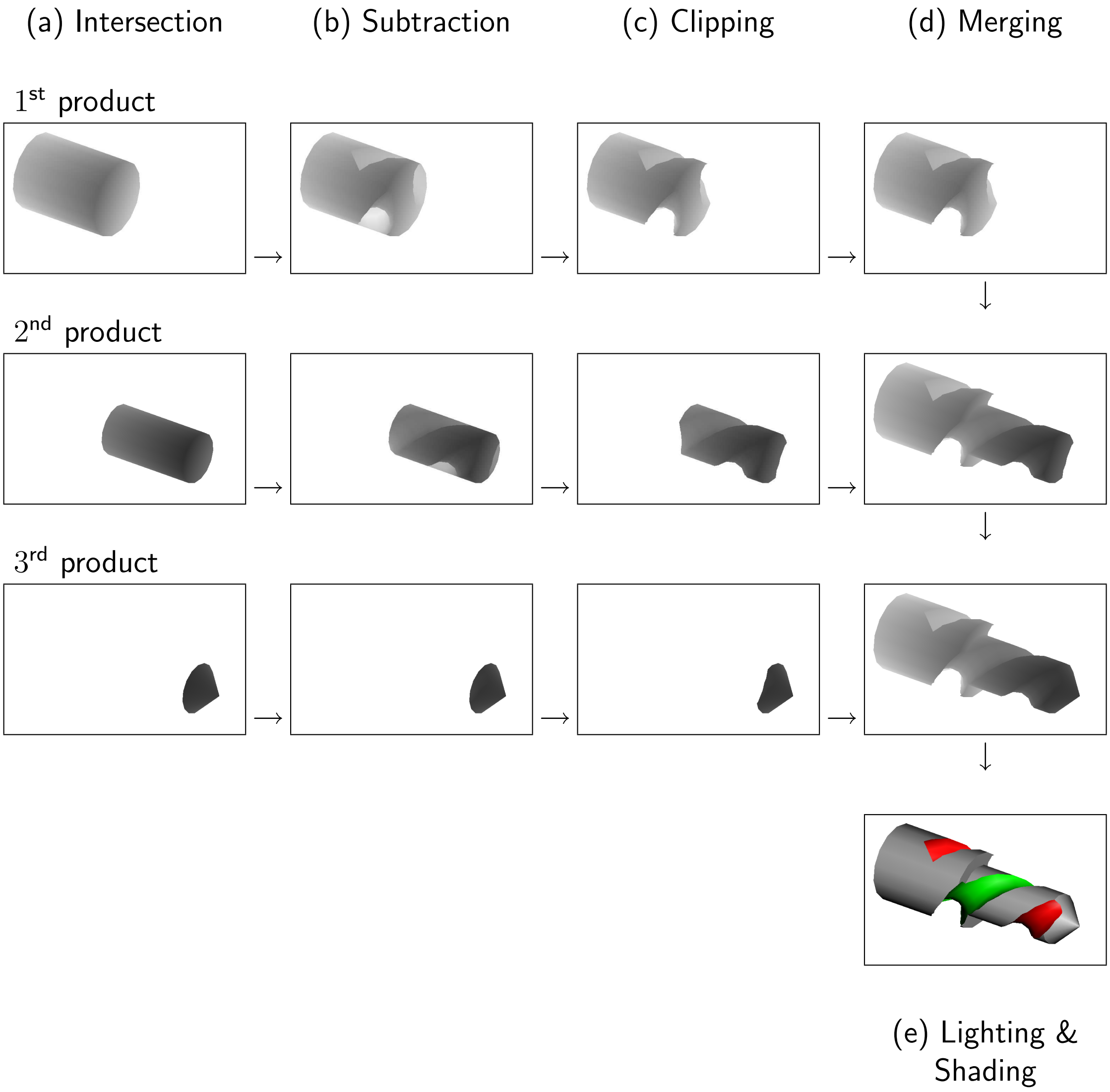


Figure 4.9: Operation of the SCS CSG Rendering Algorithm

4.4.3 Discussion

This chapter has introduced the foundational concepts of the SCS approach to CSG rendering. The CSG tree is converted into a sequence of convex subtraction steps, optionally incorporating view-dependent information. The per-pixel visible surface of the CSG tree is determined using graphics hardware implementations of convex intersection, subtraction, z-buffer clipping, merging and shading operations.

Important components of our C++ implementation of the SCS rendering algorithm appear in Appendix B and online [8]. A number of supporting routines and classes are not included for the sake of brevity, but their purpose should be clear from the context. Low-level and hardware specific issues are also discussed in the Appendix.

Chapter 4 explores theoretical aspects of the subtraction sequences used in the SCS approach. Chapter 5 considers the use of object overlap information for the reduction of subtraction sequence length. Performance aspects of our implementation of SCS and Goldfeather algorithms are the focus of Chapter 7.

Chapter 5

Permutation Embedding Sequences

"The challenge of combinatorial optimisation is to develop algorithms for which the number of elementary computational steps is acceptably small. If this challenge is not of interest to "mathematicians" it most certainly is to computer scientists. Moreover, the challenge will be met only through study of the fundamental nature of combinatory algorithms and not by any conceivable advance in computer technology." [57]
— E. Lawler

5.1 Introduction

The *Sequenced Convex Subtraction* (SCS) CSG rendering algorithm utilises *Permutation Embedding Sequences* (PESs) for z-buffer subtraction of convex objects. Sequences embedding all $n!$ permutations of n objects are $O(n^2)$ in length. This chapter focuses on several aspects of PESs including the mathematical formulation, related problems, methods of generation, shortest length sequences and techniques for efficient exhaustive computer search.

This chapter explores the notion of shortest length subtraction sequences for the SCS algorithm in order to maximise performance. A key outcome of this work is a complete set of shortest length subtraction sequences for $n \leq 5$. An efficient exhaustive approach has been unable to determine subtraction sequences shorter than $n^2 - 2n + 4$ as produced by previously reported methods [14, 35, 83]. Another outcome of this work is an $O(n^2)$ lower bound for PES length.

A PES embeds permutation sequences. A sequence is said to be embedded if it can be formed by removing entries. For example, the sequences *ab* and *ac* are embedded in *abc*. Neither *ba* or *ca* are embedded in *abc*. Subtraction sequences embedding all permutations are utilised by the SCS algorithm to ensure that all sequences of necessary subtractions are correctly handled.

Utilisation of a PES for CSG rendering poses several issues. The length of a subtraction sequence determines the number of subtraction steps and consequently, the required computation time for the SCS algorithm. The asymptotic relationship between the number of subtracted convex objects n and the PES length is $O(n^2)$. Consequentially, the time requirement for the SCS algorithm is $O(n^2)$. Another aspect is the need to construct a PES given a particular CSG tree and spatial arrangement. Several previous methods for generating PESs of length n^2, $n^2 - n + 1$ and $n^2 - 2n + 4$ are examined. Taking depth complexity into account, PESs may be generalised to *k-Permutation Embedding Sequences* (kPESs) of length kn, $kn - k + 1$ and $kn - 2k + 4$. Methods for determining a shortest length PES or kPES are examined.

To conclude this chapter, several unresolved issues and possible future directions are discussed.

This chapter is based in part on our publication *On Minimal Strings Containing the Elements of S_n by Decimation* [30].

5.1.1 Formulation

The set of n elements $\{x_1, x_2, \cdots, x_n\}$ is denoted A_n. $\mathcal{S}_n$ is the set of permutations of A_n. The number of permutations $|\mathcal{S}_n|$ is $n!$. The set of *Permutation Embedding Sequence* (PESs) is denoted $\mathcal{L}(n)$. A PES of length m is denoted L_m where $L_m \in \mathcal{L}(n)$ and $m = |L_m|$. A sequence x is embedded in (or is a subsequence of) the sequence y iff $x = \{x_1\ x_2\ \cdots\ x_n\}$ and $y = \{\ \cdots\ x_1\ \cdots\ x_2\ \cdots\ ...\ \cdots\ x_n\ \cdots\ \}$.

5.1.2 Properties of $\mathcal{L}(n)$

Several properties of PESs are of interest.

1. There are infinite PESs for finite n: $|\mathcal{L}(n)| = \infty$

   From any PES $L_m \in \mathcal{L}(n)$, it is possible to construct another PES by inserting additional entries. For example: *abca* and *abcb* both embed *abc* — and therefore embed the embedded permutations of *abc*.

2. A lower bound for the length of a PES is n: $|L_m| \geq n$

   A sequence of length n can embed at most one permutation. For example, a embeds a, but ab or ba can not embed both ab and ba.

3. A upper bound for the shortest length PES is n^2: $|L_m| \leq n^2$

   A PES formed by concatenation of n permutations embeds all permutations, since each permutation embeds all n elements. For example: *abab* embeds both *ab* and *ba* since both a and b are embedded in each copy of *ab*.

4. A PES contains all n elements at least once.

   Since a permutation consists of all n elements a sequence can only embed permutations if all n elements are present.

5. Repetition in a PES is redundant: $L_m = \{\ \cdots\ x\ x\ \cdots\ \} \rightarrow L_{m-1} = \{\ \cdots\ x\ \cdots\ \}$

   A repetition of an entry in a PES is unnecessary since no such repetition occurs in any permutation. For example: *abba* embeds the same permutations as *aba* — *ab* and *ba*.

6. PES in normalised form: $\{\ c\ b\ a\ \} \rightarrow \{\ a\ b\ c\ \}$

   Each sequence of n elements has $n! - 1$ alternative representations by relabelling. For example: *abc*, *acb*, *bac*, *bca*, *cab* and *cba* can all be relabelled to *abc*. Sequences expressed in normalised form are considered representative of the $n! - 1$ alternatives.

5.1.3 Shortest Length PES Examples

Shortest length PESs for $n = 2$ and $n = 3$ are provided here as examples. Removed elements are denoted '$\star$'.

$$\begin{array}{rlrl}
n = & 2 & n = & 3 \\
 & & & \\
L_3 = & aba & L_7 = & abcacba \\
 & & & \\
ab \rightarrow & ab\star & abc \rightarrow & abc\star\star\star\star \\
ba \rightarrow & \star ba & acb \rightarrow & a\star c\star\star b\star \\
 & & bca \rightarrow & \star bca\star\star\star \\
 & & bac \rightarrow & \star b\star ac\star\star \\
 & & cab \rightarrow & \star\star ca\star b\star \\
 & & cba \rightarrow & \star\star c\star\star ba
\end{array}$$

$\therefore \mathcal{S}_2$ are embedded in *aba* $\qquad\qquad$ $\therefore \mathcal{S}_3$ are embedded in *abcacba*

No shorter sequences have been found by exhaustive automated search. Additional shortest length PESs are discussed in Section 5.4.8 and appear in Table 5.7.

5.2 Related Mathematical Problems

In relation to Permutation Embedding Sequences, we introduce related mathematical problems such as de Bruijn Sequences [21] and Ouroborean Rings [86]. These sequences are similar to PESs except that permutations are contiguously embedded. These sequences are not useful as CSG subtraction sequences due to their excessive length. We discuss them for the purpose of placing PESs in a broader mathematical context.

5.2.1 de Bruijn Sequences

A binary *de Bruijn sequence* is a periodic binary sequence with a period of 2^k bits $a_1 a_2 \cdots a_{2^k}$, such that each binary string of size k is somewhere in the sequence contiguously. Generalising, n-ary de Bruijn sequences with a period of n^k contain each sequence of n objects of length k contiguously. A cycle of several de Bruijn sequences is given in Table 5.1.

Definition: *Suppose n and k are positive integers. An n-ary de Bruijn sequence of span k is a n-ary cycle of period n^k containing n^k distinct k-tuples in each cycle. Equivalently, every possible n-ary k-tuple occurs precisely once in a period of the n-ary de Bruijn sequence of span k.*[21]

n	k	de Bruijn sequence cycle	length
2	1	*ab*	2
2	2	*aabb*	4
2	3	*aaababbb*	8
2	4	*aaaabaabbababbbb*	16
3	1	*abc*	3
3	2	*aabacbbcc*	9
3	3	*aaabaacabbabcacbaccbbbcbccc*	27
4	1	*abcd*	4
4	2	*aabacadbbcbdccdd*	16
4	3	*aaabaacaadabbabcabdacbaccacdadbadcaddbbbcbbdbccbcdbdcbddcccdcddd*	64
5	1	*abcde*	5
5	2	*aabacadaebbcbdbeccdceddee*	25

Table 5.1: de Bruijn Sequences

The embedding of n-ary k-tuple in a de Bruijn sequence for $n = 3$, $k = 2$ is demonstrated here as an example.

$$\text{sequence} = \quad aabacbbcc\ aabacbbcc\ ..$$

$$\begin{aligned}
aa \rightarrow &\quad aa\star\star\star\star\star\star\star\ .. \\
ab \rightarrow &\quad \star ab\star\star\star\star\star\star\ .. \\
ba \rightarrow &\quad \star\star ba\star\star\star\star\star\ .. \\
ac \rightarrow &\quad \star\star\star ac\star\star\star\star\ .. \\
cb \rightarrow &\quad \star\star\star\star cb\star\star\star\ .. \\
bb \rightarrow &\quad \star\star\star\star\star bb\star\star\ .. \\
bc \rightarrow &\quad \star\star\star\star\star\star bc\star\ .. \\
cc \rightarrow &\quad \star\star\star\star\star\star\star cc\ .. \\
ca \rightarrow &\quad \star\star\star\star\star\star\star\star c\ a..
\end{aligned}$$

5.2.2 Ouroborean Rings

Sequences of minimal length containing every possible n-ary k-tuple contiguously have been called *Ouroborean Rings* by Ian Stewart [86]. The word Ourobouros comes from an ancient Egyptian representation of a snake swallowing its tail and has a particular meaning in ancient alchemical works. The first algorithm to generate an Ouroborean ring was offered by Flye-Sainte Marie in the 19th century [60]. These sequences are also known as *full length nonlinear shift register cycles* [34].

An Ouroborean ring is constructed from a de Bruijn sequence. The Ouroborean ring is a de Bruijn sequence cycle, followed by the first $k-1$ entries in the de Bruijn sequence [61, 39, 55]. The length of the Ouroborean ring is therefore $n^k + k - 1$. Several example Ouroborean rings are listed in Table 5.2.

n	k	de Bruijn sequence cycle	Ouroborean ring
2	2	*abba*	*abba a*
2	3	*abbbabaa*	*abbbabaa ab*
3	2	*aabacbbcc*	*aabacbbcc a*
3	3	*aaabbbcccbcbbaccacbabcabaac*	*aaabbbcccbcbbaccacbabcabaac aa*
4	2	*aabacadbbcbdccdd*	*aabacadbbcbdccdd a*

Table 5.2: Ouroborean Rings

An important application of these sequences in cryptography is the generation of pseudo-random binary sequences of maximal length [38, 12]. Other applications have been described [34, 85, 86].

5.2.3 Baltimore Hilton Inn Problem

Ouroborean Rings can be applied to the cryptographic problem of the *Baltimore Hilton Inn* [34, 41]. The problem is concerned with a cipher lock system using a 4 digit decimal code. There are $10,000$ possible codes for the lock. A naive attack on the lock, trying each possible code in turn, requires $40,000$ keypresses at most. Since the lock opens whenever the most recent 4 digits match the code, an Ouroborean sequence can be used instead — requiring only 10,003 keypresses.

5.2.4 Ouroborean Ring PESs

Since Ouroborean rings contain all k-tuples contigously, they are also embedding sequences for k-permutations. The length of these PESs can be optimised somewhat by removing immediately repeated entries. (Property 5.1.2.5)

Figure 5.1 illustrates Ouroborean rings condensed into PESs for $n = 2$ and $n = 3$. As it happens, *aba* is a shortest length PES for $n = 2$, but *abcbcbacacbabcabaca* is not shortest for $n = 3$. A shortest length PES for $n = 3$ is *abcabac*.

Ouroborean rings are permutation embedding sequences, of $O(n^n)$ length in general. Although they could be used by the SCS CSG rendering algorithm, $O(n^2)$ length PESs are preferable due to their shorter length. In CSG rendering, contiguous embedding of permutations is unnecessary for correct handling of subtractions.

n,k	**Ouroborean Ring**	**Permutation Embedding Sequence**
2	*abbaa*	*aba*
3	*aaabbbcccbcbbaccacbabcabaacaa*	*abcbcbacacbabcabaca*

Figure 5.1: Ouroborean Ring Permutation Embedding Sequences

5.3 PES Generation

This section focuses on the problem of generating PESs given a number of symbols n. We describe a simple method for determining sequences of length $n^2 - n + 1$. More sophisticated methods [56, 14, 35] can be used to obtain sequences of length $n^2 - 2n + 4$ for $n \geq 4$. In the context of CSG rendering we are also interested in kPESs — sequences embedding all k-permutations of n elements. By determining the depth complexity k a kPES can be used rather than a PES, resulting in shorter subtraction sequences and faster rendering. Depth complexity was discussed previously in Section 3.4.1 and Section 4.3.2.

5.3.1 PESs of Length n^2

A simple way to generate permutation embedding sequences is by concatenating n permutations s_i, where $s_i \in \mathcal{S}_n$ and $1 \leq i \leq n$. Denoting '$\cdot$' as the concatenation operator: $s_1 \cdot s_2 \cdot ... \cdot s_n$ is a permutation embedding sequence. Each s_i is a permutation of n elements, therefore embedding all n elements. The j^{th} entry of every permutation is therefore embedded in s_j. Since $1 \leq i \leq n$ and $1 \leq j \leq n$, these sequences are PESs. The length of these PESs is n^2 since the length of s_i is n and there are n of them concatenated together. Generalising to k-permutations results in kPESs of length kn: $s_1 \cdot s_2 \cdot ... \cdot s_k$.

5.3.2 PESs of Length $n^2 - n + 1$

The previous method can be refined by using Property 5.1.2.5, removing immediately repeated entries. The sequence of permutations, s_i for $1 \leq i \leq n$ can be chosen so that neighbouring permutations share the same element.

For example, s_x and s_y can be chosen such that s_y is the reversal of s_x. The embedding sequence is formed by alternating between s_x and s_y, and removing the repeated entry at each boundary. That results in $n-1$ entries being removed and a PES of length n^2-n+1.

- $n=2$, $s_x=ab$ and $s_y=ba$

  $s_x \cdot s_y \rightarrow ab \ \cdot \ ba \rightarrow aba$

- $n=3$, $s_x=abc$ and $s_y=cba$

  $s_x \cdot s_y \cdot s_x \rightarrow abc \ \cdot \ cba \ \cdot \ abc \rightarrow abcbabc$

Generalising to k-permutations results in kPESs of length $kn-k+1$:

- $n=4$, $k=2$, $s_x=abcd$ and $s_y=dcba$

  $s_x \cdot s_y \rightarrow abcd \ \cdot \ dcba \rightarrow abcdcba$

- $n=4$, $k=3$, $s_x=abcd$ and $s_y=dcba$

  $s_x \cdot s_y \cdot s_x \rightarrow abcd \ \cdot \ dcba \ \cdot \ abcd \rightarrow abcdcbabcd$

5.3.3 Adleman Sequence of Length n^2-2n+4

A three step method for constructing a PES of length n^2-2n+4 was given by Adleman [14] in 1974.

1. Begin with a sequence of length n^2-3n+4 by repeating the sequence $x_1x_2\cdots x_{n-1}$.
2. Insert the n^{th} symbol x_n after the i^{th} occurrence of the symbol x_{n-i} for all i, $1 \leq i \leq n-2$.
3. Insert the n^{th} symbol x_n at the beginning and end of the sequence.

For example:

- $n=3$

  1. *abab*
  2. *abcab*
  3. *cabcabc*

$$\begin{array}{ll} \star abc\star\star\star & \star a\star c\star b\star \\ \star\star b\star a\star c & \star\star bca\star\star \\ cab\star\star\star\star & c\star b\star a\star\star \end{array}$$

$$\therefore \mathcal{S}_3 \text{ are embedded in } cabcabc$$

- $n = 4$

  1. *abcabcab*
  2. *abcdabdcab*
  3. *dabcdabdcabd*

$$\begin{array}{llll}
\star abcd\star\star\star\star\star\star\star & \star\star b\star\star a\star\star c\star\star d & \star\star\star c\star abd\star\star\star\star & dabc\star\star\star\star\star\star\star\star \\
\star ab\star d\star\star\star c\star\star\star & \star\star b\star\star a\star dc\star\star\star & \star\star\star c\star a\star d\star\star b\star & da\star c\star\star b\star\star\star\star\star \\
\star a\star c\star\star bd\star\star\star\star & \star\star bc\star a\star d\star\star\star\star & \star\star\star c\star\star b\star\star a\star d & d\star b\star\star a\star\star c\star\star\star \\
\star a\star cd\star b\star\star\star\star\star & \star\star bcda\star\star\star\star\star\star & \star\star\star c\star\star bd\star a\star\star & d\star bc\star a\star\star\star\star\star\star \\
\star a\star\star d\star b\star c\star\star\star & \star\star b\star da\star\star c\star\star\star & \star\star\star cdab\star\star\star\star\star & d\star\star c\star ab\star\star\star\star\star \\
\star a\star\star d\star\star\star c\star b\star & \star\star b\star d\star\star\star ca\star\star & \star\star\star cd\star b\star\star a\star\star & d\star\star c\star\star b\star\star a\star\star
\end{array}$$

$$\therefore \mathcal{S}_4 \text{ are embedded in } dabcdabdcabd$$

These sequences are PESs. A proof by induction for $n \geq 3$ is provided in the original paper [14]. Our C++ implementation of the algorithm is included in Appendix B.4.1.

An alternative formulation resulting in essentially the same sequences was described by Koutas and Hu [56] in 1975.

5.3.4 Galbiati Sequence of Length $n^2 - 2n + 4$

Another method for constructing a PES of length $n^2 - 2n + 4$ was given in 1976 by Galbiati and Preparata [35].

1. The n elements are partitioned into two sets: $\{ a, b, c, d \}$ and $U = A_n - \{ a, b, c, d \}$.
2. U also denotes an arbitrary permutation of the elements of U.
3. The sequences $bcdbcd\cdots$ and $aUaU\cdots$ are interleaved in the following manner:

$$\begin{array}{cccc|cccc|cccc|cccc|cccc|cc}
 & b & & c & & d & & b & & c & & d & & b & & c & & d & & b & & \cdots \\
 & & a & & U & & a & & U & & a & & U & & a & & U & & a & & U & \cdots \\
\hline
\longrightarrow & b & a & c & U & d & a & b & U & c & a & d & U & b & a & c & U & d & a & b & U & \cdots \\
 & & & & \multicolumn{4}{c|}{\text{Segment 1}} & \multicolumn{4}{c|}{\text{Segment 2}} & \multicolumn{4}{c|}{\text{Segment 3}} & \multicolumn{4}{c|}{\text{Segment 4}} & & \cdots
\end{array}$$

The combined sequence contains a prefix of three elements and $n - 1$ segments of length $n - 1$. The total length is therefore $3 + (n - 1)^2 = n^2 - 2n + 4$.

For example:

- $n = 4$

  $\{\ a, b, c, d\ \}$ and $U = \{\ \}$

  $bac|dab|cad|bac$

  $bacdabcadbac$

$$\begin{array}{llll}
\star a\star\star\star bc\star d\star\star\star & bacd\star\star\star\star\star\star\star\star & \star\star c\star ab\star\star d\star\star\star & \star\star\star dabc\star\star\star\star\star \\
\star a\star\star\star b\star\star d\star\star c & ba\star d\star\star c\star\star\star\star\star & \star\star c\star a\star\star\star d\star\star c & \star\star\star da\star c\star\star b\star\star \\
\star ac\star\star b\star\star d\star\star\star & b\star c\star a\star\star\star d\star\star\star & \star\star c\star\star b\star ad\star\star\star & \star\star\star d\star b\star a\star\star\star c \\
\star acd\star b\star\star\star\star\star\star & b\star cda\star\star\star\star\star\star\star & \star\star c\star\star b\star\star d\star a\star & \star\star\star d\star bca\star\star\star\star \\
\star a\star d\star bc\star\star\star\star\star & b\star\star da\star c\star\star\star\star\star & \star\star cdab\star\star\star\star\star\star & \star\star\star d\star\star ca\star b\star\star \\
\star a\star d\star\star c\star\star b\star\star & b\star\star d\star\star ca\star\star\star\star & \star\star cd\star b\star a\star\star\star\star & \star\star\star d\star\star c\star\star ba\star
\end{array}$$

$$\therefore \mathcal{S}_4 \text{ are embedded in } bacdabcadbac$$

- $n = 5$

  $\{\ a, b, c, d\ \}$ and $U = \{\ e\ \}$

  $bac|edab|ecad|ebac|edab$

  $bacedabecadebacedab$

A proof by induction that these sequences are PES for $n \geq 4$ is provided in the original paper [35]. It follows from the proof that the first $kn-k+1$ elements are kPESs. However, this is no shorter than the sequences previously formulated in Section 5.3.2, which are known not to be shortest length in general. The shortest length kPES given in Table 5.8 for $n = 5$, $k = 4$ is 16 while $kn - k + 1 = 17$. The next section discusses kPES sequences of length $kn - 2k + 4$.

A C++ implementation of the Galbiati PES algorithm is included in Appendix B.4.2.

5.3.5 Savage Sequence of Length $kn - 2k + 4$

The first paper to focus on sequences embedding all k-permutations (kPESs) was by Savage [83] in 1982. A method of construction similar to Adleman [14] results in sequences of length $kn - 2k + 4$ for $3 \leq k \leq n$ and in the special case that $k = n$, the method is essentially equivalent to Adleman, resulting in sequences of length $n^2 - 2n + 4$.

1. The n elements are partitioned into two sets: $\alpha = \{ a, b, \cdots, x_{k-1}\}$ and $\beta = \{x_k, \cdots, x_n\}$.
2. α forms a recurring sequence $A = ab \cdots x_{k-1} ab \cdots$ of length $k^2 - 3k + 4$.
3. β forms a sequence $B = x_k \cdots x_n$.
4. Insert B after the i^{th} occurance of the $(k-i)^{\text{th}}$ symbol x_{k-i} in A for all i, $1 \leq i \leq k-2$.
5. Insert B at the beginning and end of the sequence.

For example:

- $n = 3, k = 3$

    1. $\alpha = \{ a, b \}$, $\beta = \{ c \}$
    2. $A = abab$
    3. $B = c$
    4. $abcab$
    5. $cabcabc$

    The sequence $cabcabc$ is the same sequence produced by the Adleman method demonstrated for $n = 3$ in Section 5.3.3.

- $n = 4, k = 3$

  1. $\alpha = \{\ a, b\ \}$, $\beta = \{\ c, d\ \}$
  2. $A = abab$
  3. $B = cd$
  4. $abcdab$
  5. $cdabcdabcd$

$$\begin{array}{llllll}
\star\star abc\star\star\star\star\star & \star\star ab\star d\star\star\star\star & \star\star a\star c\star\star b\star\star & \star\star a\star cd\star\star\star\star & \star\star a\star\star d\star b\star\star & \star\star a\star\star d\star\star c\star \\
\star\star\star b\star\star a\star c\star & \star\star\star b\star\star a\star\star d & \star\star\star bc\star a\star\star\star & \star\star\star bcd\star\star\star\star & \star\star\star b\star da\star\star\star & \star\star\star b\star d\star\star c\star \\
c\star ab\star\star\star\star\star\star & c\star a\star c\star\star\star\star\star & c\star\star b\star\star a\star\star\star & c\star\star b\star d\star\star\star\star & cda\star\star\star\star\star\star\star & cd\star b\star\star\star\star\star\star \\
\star dab\star\star\star\star\star\star & \star da\star c\star\star\star\star\star & \star d\star b\star\star a\star\star\star & \star d\star bc\star\star\star\star\star & \star d\star\star c\star a\star\star\star & \star d\star\star c\star\star b\star\star
\end{array}$$

5.4 Shortest Length PES

In previous sections several PES encoding methods have been described. These include the Galbiati [35] sequences of length $n^2 - 2n + 4$ and Savage [83] sequences of length $kn - 2k + 4$. Although there are no known methods for producing a PES of shorter length, no proof of optimality has been given. It is not known if PESs of shorter length exist, or can be constructed systematically. This section examines the prospect of determining the shortest possible permutation embedding sequences.

First, several foundational concepts are formulated which aim to minimise the time required by a computer to exhaustively search for PESs or kPESs. A full set of shortest length PESs is presented for computationally manageable configurations. The results indicate that the Adleman [14], Galbiati [35] and Savage [83] methods are shortest length for $n \leq 5$, since exhaustive search has not resulted in the finding of any sequences of shorter length. Therefore, this work adds support to the understanding of SCS Subtraction sequences being $O(n^2)$ or $O(kn)$ in length, as implied by previous work.

5.4.1 Brute Force Search

Brute force search (or exhaustive search) is a very general purpose computer science problem solving approach. It involves generating each potential solution candidate and checking whether the candidate is satisfactory. In this manner a solution will always be found, if it exists. It also has the advantage of being simple to implement. The cost of this approach is proportional to the number of candidate solutions, which can grow rapidly as the size of the problem increases. One approach to resolving this issue of "combinatorial explosion" is to reduce the search space size, in the hope of making it practical for exhaustive search. The search space size could be reduced by excluding unsatisfactory candidates without generating or testing them. The search space size could also be reduced by consolidating groups of candidate solutions into representative candidates by means of equivalence or symmetry. Another approach to the search space size problem is to test the more promising candidate solutions first, in the hope of a solution being found early.

Our approach in this work is to use exhaustive search of sequences to determine if PESs or kPESs exist (or do not exist) of a particular length. We reduce the search space significantly by using *Normalised No Repeat* (NNR) sequences, rather than all n^l sequences of a particular length l. The cost of testing each candidate sequence is proportional to the number of embedded permutations: $O(n!)$ or $O(\frac{n!}{(n-k)!})$. A brute force approach without using NNR sequences would require $O(n^l n!)$ or $O(\frac{n^l n!}{(n-k)!})$ time.

For example, for $n = 5$, $l = 19$ there are $n^l = 5^{19} = 19,073,486,328,125$ candidate sequences. The NNR search space is only $2,798,806,985$ — approximately seven thousand times smaller.

5.4.2 Normalised No Repeat (NNR) Sequences

The size of the search space can be reduced by constraining sequences to those having known characteristics of shortest length solutions. Properties 4, 5 and 6 from Section 5.1.2 can be combined to form what we call the *Normalised No Repeat* (NNR) search space. NNR constrains sequences to those:

- Containing all n elements at least once — according to Property 4 a sequence cannot be permutation embedding without containing all n elements.
- Containing no adjacent repetitions of elements — according to Property 5 no shortest length PES will contain adjacent repetition of an element.
- We call *normalised* sequences — according to Property 6. Normalised sequences have the first a somewhere to the left of the first b, the first b somewhere to the left of the first c, and so on. Normalised sequences are representative of $n! - 1$ other equivalent sequences that do not need to be considered for the purpose of exhaustive search.

5.4.3 NNR Generation

For the purpose of exhaustive search of candidate sequences it is necessary to determine all NNR sequences of a particular length. The permutation embedding properties of each NNR sequence is tested in turn.

A recursive algorithm for generating the NNR search space is given in Algorithm 5.1. The NNR sequence is constructed from left to right by appending entries from either the *active* set or *waiting* set. The *active* set contains elements already present in the sequence, while the *waiting* set contains elements not yet in the sequence but which must appear at least once. Initially the active set is empty and the waiting set contains all n elements, in order. Elements are moved from the waiting set to the active set as the algorithm proceeds.

Algorithm 5.1 NNR Generation

```
Require: $n$ is the number of elements
Require: $l$ is the sequence length

  $active \leftarrow \{\ \}$
  $waiting \leftarrow \{\ a, b, c, \cdots, x_n\}$
  $sequence \leftarrow \{\ \}$

  if $|sequence| = l$ then
    output $sequence$
  else

    if $|waiting| > 0$ then
      $x \leftarrow$ left-most element in $waiting$
      return generate( $active \cup x$ , $waiting - x$ , $sequence \cdot x$ )

    if $|active| > 0$ and $|sequence| + |waiting| < l$ then
      for all $x \in active$ do
        if $x \neq$ right-most element in $sequence$ then
          return generate( $active$ , $waiting$ , $sequence \cdot x$ )
```

The algorithm is recursive with a base case and two recursive cases. In the base case, if the sequence is of sufficient length it is output by the algorithm. In the first recursive case, the next item from the waiting set is appended to the sequence and moved to the active set. In the second recursive case, each item in the active set is appended to the sequence, except the current right-most element of the sequence. The active set is only used while there remains enough space in the sequence for the remaining waiting items.

In this manner the following properties are ensured:

- All n elements are present in the sequence. (Section 5.1.2, Property 4)
- There is no adjacent repetition of any element. (Section 5.1.2, Property 5)
- The sequence is normalised. (Section 5.1.2, Property 6)

The complete NNR search space for several example configurations is presented in Table 5.3, with permutation embedding NNR sequences appearing in bold. For one element there is only one possible NNR sequence: a. For $n = 1$, $l > 1$ it is impossible to form an NNR sequence without repeating a. For two elements there is one NNR sequence for $l \geq 2$ formed by alternating a and b, with aba the shortest length PES. For three elements there are seven shortest length PESs of the thirty one NNR sequences of length seven. For four elements there are no PESs for NNR sequences of length four, five or six. As shown in following sections, the shortest length PES for $n = 4$ is $l = 12$ in a NNR search space of $28,501$ sequences. For five elements, the shortest length PES is $l = 19$ in a NNR search space of $2,798,806,985$ sequences.

The number of NNR sequences $|\Omega|$, and the total number of sequences are given in the rightmost columns. It can be observed that the number of possible NNR sequences of a particular length is substantially less than n^l. In practice, it is much more computationally efficient to generate and test each NNR sequence than all possible sequences of the same length.

The following sections formulate a way of calculating the number of NNR sequences of a particular length, and from this, a method for determining individual NNR sequences in a non-recursive manner.

n	**l**	Ω	|Ω|	n^l
1	1	**a**	1	1
2	2	ab	1	4
2	3	**aba**	1	8
2	4	**abab**	1	16
2	5	**ababa**	1	32
3	3	abc	1	27
3	4	abca abcb abac	3	81
3	5	abcab abcac abcba abcbc abaca abacb ababc	7	243
3	6	abcaba abcabc abcaca abcacb abcbab abcbac abcbca abcbcb abacab abacac abacba abacbc ababca ababcb ababac	15	729
3	7	abcabab **abcabac** **abcabca** abcabcb abcacab abcacac **abcacba** abcacbc abcbaba **abcbabc** abcbaca **abcbacb** abcbcab abcbcac abcbcba abcbcbc **abacaba** abacabc abacaca abacacb **abacbab** abacbac abacbca abacbcb ababcab ababcac ababcba ababcbc abababaca ababacb abababc	31	2187
4	4	abcd	1	256
4	5	abcda abcdb abcdc abcad abcbd abacd	6	1024
4	6	abcdab abcdac abcdad abcdba abcdbc abcdbd abcdca abcdcb abcdcd abcada abcadb abcadc abcabd abcacd abcbda abcbdb abcbdc abcbad abcbcd abacda abacdb abacdc abacad abacbd ababcd	25	4096
5	5	abcde	1	3125
5	6	abcdea abcdeb abcdec abcded abcdae abcdbe abcdce abcade abcbde abacde	10	15625
5	7	abcdeab abcdebc abcdecd abcdede abcdeac abcdebd abcdece abcdeda abcdead abcdebe abcdeca abcdedb abcdeae abcdeba abcdecb abcdedc abcdaea abcdbea abcdcea abcdaeb abcdbeb abcdceb abcdaec abcdbec abcdcec abcdaed abcdbed abcdced abcdabe abcdbce abcdcde abcdace abcdbde abcdcae abcdade abcdbae abcdcbe abcadea abcbdea abcadeb abcbdeb abcadec abcbdec abcaded abcbded abcadae abcbdae abcadbe abcbdbe abcadce abcbdce abcabde abcbcde abcacde abcbade abacdea abacdeb abacdec abacded abacdae abacdbe abacdce abacade abacbde ababcde	65	78125

Permutation embedding sequences appear in **bold**.

Table 5.3: Example NNR Sequences

5.4.4 NNR Size

The size of an NNR search space Ω, denoted $|\Omega|$ can be represented with the following recursive definition, denoting n as the number of elements and l as the length of the sequence:

$$|\Omega(n,l)| = (n-1) \times |\Omega(n,l-1)| + |\Omega(n-1,l-1)| \tag{5.1}$$

With the following base cases:

$$n = 1,\ l = 1 \ \rightarrow |\Omega| = 1 \tag{5.2}$$

$$n > 1,\ l = 1 \ \rightarrow |\Omega| = 0 \tag{5.3}$$

$$n = 1,\ l > 1 \ \rightarrow |\Omega| = 0 \tag{5.4}$$

It follows that:

$$l = n \ \rightarrow |\Omega| = 1 \tag{5.5}$$

$$l < n \ \rightarrow |\Omega| = 0 \tag{5.6}$$

The two recursive cases in Equation 5.1 are as follows: the number of NNR sequences of length $l-1$ times the $n-1$ possible ending entries, and the number of NNR sequences of length $l-1$ of $n-1$ entries with the n^{th} entry appended.

The size of various NNR search spaces $|\Omega|$ is provided in Table 5.4, with Ω containing shortest length PESs appearing in bold. For shortest length PESs, $|\Omega|$ increases rapidly: $1, 1, 31, 28501, 2798806985$ for $n = 1, 2, 3, 4, 5$. As a fraction of the total number of sequences: $\frac{|\Omega|}{n^l} = 1$, 0.125, 0.014, 0.0017, 0.00015. In the context of exhaustive search this indicates that $|\Omega|$ is a decreasing fraction as n and l increase: $\lim\limits_{n,l\rightarrow\infty} \frac{|\Omega|}{n^l} = 0$.

Algorithm 5.2 is an efficient algorithmic approach to calculating the size of an NNR search space. The algorithm requires $O(nl)$ time and $O(n)$ storage. In effect, a subset of the complete NNR size table is computed (as in Table 5.4) while storing only two rows in memory at a time. A C++ implementation appears in Appendix B.5.1. An arbitrary precision `Integer` class is used for intermediate values and output, since the number of possible NNR sequences can become extremely large for relatively small n and l. As an example, for $n = 7$, $l = 21$, $|\Omega| = 4,306,078,895,384$ requiring at least 42 bits of storage.

	n						
l	**1**	2	3	4	5	6	7
1	1						
2		1					
3		**1**	1				
4		1	3	1			
5		1	7	6	1		
6		1	15	25	10	1	
7		1	**31**	90	65	15	1
8		1	63	301	350	140	21
9		1	127	966	1,701	1,050	266
10		1	255	3,025	7,770	6,951	2,646
11		1	511	9,330	34,105	42,525	22,827
12		1	1,023	**28,501**	145,750	246,730	179,487
13		1	2,047	86,526	611,501	1,379,400	1,323,652
14		1	4,095	261,625	2,532,530	7,508,501	9,321,312
15		1	8,191	788,970	10,391,745	40,075,035	63,436,373
16		1	16,383	2,375,101	42,355,950	210,766,920	420,693,273
17		1	32,767	7,141,686	171,798,901	1,096,190,550	2,734,926,558
18		1	65,535	21,457,825	694,337,290	5,652,751,651	17,505,749,898
19		1	131,071	64,439,010	**2,798,806,985**	28,958,095,545	110,687,251,039
20		1	262,143	193,448,101	11,259,666,950	147,589,284,710	693,081,601,779
21		1	524,287	580,606,446	45,232,115,901	749,206,090,500	4,306,078,895,384

Ω containing shortest length PESs appear in **bold**.

Table 5.4: NNR Size $|\Omega|$

Algorithm 5.2 NNR Size $|\Omega|$

```
Require: $n$ is the number of elements
Require: $l$ is the sequence length
  if $l < n$ or $n < 1$ or $l < 1$ then
    return 0
  else
    prev $\leftarrow \{ 1, 0, 0, \cdots \}$
    next $\leftarrow \{\}$
    for all $i \in [1, l)$ do
      next$_0 \leftarrow 0$
      for all $j \in [1, n)$ do
        next$_j \leftarrow$ prev$_j \times j +$ prev$_{j-1}$
      prev $\leftarrow$ next
    return prev$_{n-1}$
```

5.4.5 NNR Partitions

The previous section examined the computation of NNR search space size $|\Omega|$ given n and l. This section is concerned with NNR sequence generation based on an index $i_\Omega \in [0, |\Omega|)$, as an alternative to the recursive approach previously discussed in Section 5.4.3. Each NNR sequence can be classified as belonging to a particular NNR partition $\gamma \in \Gamma$. The number of partitions $|\Gamma|$ and the size of each partition $|\gamma|$ can be computed given n and l. For a particular partition γ each NNR sequence can be efficiently generated based on an integer index $i_\gamma \in [0, |\gamma|)$.

NNR partitioning provides an integer basis for representing NNR sequences. A subset of the NNR search space $|\Omega|$ can be specified in terms of a set of integer indexes $i_\Omega \in [0, |\Omega|)$. For the purpose of exhaustive search this can be used for distributing the NNR search space across multiple CPUs or testing candidate NNR sequences in a specific order.

Another advantage of NNR partitioning is that specific partitions can be exhaustively searched without any sequences in other partitions being generated or tested. NNR partitioning also leads to a proof of $O(n^2)$ shortest length PES, as discussed in the subsequent section.

An NNR sequence can be subdivided into n segments $s_1 \cdot s_2 \cdot \cdots \cdot s_n$ according to the position of the first occurrence of each element. As an example, the PES *abacbab* is subdivided into: $s_1 = a$, $s_2 = ba$, $s_3 = cbab$. The length of each segment $|s_i|$ where $i \in [1, n]$ must be greater or equal to one, and the sum of the segment lengths equals the NNR sequence length l. The first partition s_1 is always a to maintain the no-repeat property of NNR sequences. Formally:

$$|s_1| + |s_2| + \cdots + |s_n| = l \tag{5.7}$$

with:

$$s_1 = a,\ |s_2| \geq 1,\ \cdots,\ |s_n| \geq 1 \tag{5.8}$$

Each combination of $|s_1|, |s_2|, \cdots, |s_n|$ we call an NNR *partition* denoted γ for a given n and l. The set of partitions for n and l is denoted Γ. As an example for $n = 3$ and $l = 7$ there are five partitions:

$\lvert s_1\rvert, \lvert s_2\rvert, \lvert s_3\rvert$	$\lvert s_1\rvert, \lvert s_2\rvert, \lvert s_3\rvert$	$\lvert s_1\rvert, \lvert s_2\rvert, \lvert s_3\rvert$	$\lvert s_1\rvert, \lvert s_2\rvert, \lvert s_3\rvert$	$\lvert s_1\rvert, \lvert s_2\rvert, \lvert s_3\rvert$
1, 5, 1	1, 4, 2	1, 3, 3	1, 2, 4	1, 1, 5
$ab\star\star\star\star c$	$ab\star\star\star c\star$	$ab\star\star c\star\star$	$ab\star c\star\star\star$	$abc\star\star\star\star$

As further examples NNR partitions for $n = 4$ and $n = 5$ are given in Table 5.5. Each partition γ and the number of NNR sequences in the partition $|\gamma|$ is given for each combination of n and l. Entries that vary for different NNR sequences of the same partition are denoted "$\star$". The NNR sequences Ω is the set of NNR sequences for all partitions Γ. Note that the listed NNR sizes $|\Omega|$ correspond to those in Table 5.4 given in the previous section concerned with direct computation of $|\Omega|$.

n	**l**	$\|\Gamma\|$	γ	$\|\gamma\|$	$\|\Omega\|$
4	4	1	$abcd$	1	1
4	5	3	$ab\star cd$	1	6
			$abc\star d$	2	
			$abcd\star$	3	
4	6	6	$ab\star\star cd$	1	25
			$ab\star c\star d$	2	
			$ab\star cd\star$	3	
			$abc\star\star d$	4	
			$abc\star d\star$	6	
			$abcd\star\star$	9	
4	7	10	$ab\star\star\star cd$	1	90
			$ab\star\star c\star d$	2	
			$ab\star\star cd\star$	3	
			$ab\star c\star\star d$	4	
			$ab\star c\star d\star$	6	
			$ab\star cd\star\star$	9	
			$abc\star\star\star d$	8	
			$abc\star\star d\star$	12	
			$abc\star d\star\star$	18	
			$abcd\star\star\star$	27	
4	8	15	$ab\star\star\star\star cd$	1	301
			$ab\star\star\star c\star d$	2	
			$ab\star\star\star cd\star$	3	
			$ab\star\star c\star\star d$	4	
			$ab\star\star c\star d\star$	6	
			$ab\star\star cd\star\star$	9	
			$ab\star c\star\star\star d$	8	
			$ab\star c\star\star d\star$	12	
			$ab\star c\star d\star\star$	18	
			$ab\star cd\star\star\star$	27	
			$abc\star\star\star\star d$	16	
			$abc\star\star\star d\star$	24	
			$abc\star\star d\star\star$	36	
			$abc\star d\star\star\star$	54	
			$abcd\star\star\star\star$	81	

n	**l**	$\|\Gamma\|$	γ	$\|\gamma\|$	$\|\Omega\|$
5	5	1	$abcde$	1	1
5	6	4	$ab\star cde$	1	10
			$abc\star de$	2	
			$abcd\star e$	3	
			$abcde\star$	4	
5	7	10	$ab\star\star cde$	1	65
			$ab\star c\star de$	2	
			$ab\star cd\star e$	3	
			$ab\star cde\star$	4	
			$abc\star\star de$	4	
			$abc\star d\star e$	6	
			$abc\star de\star$	8	
			$abcd\star\star e$	9	
			$abcd\star e\star$	12	
			$abcde\star\star$	16	
5	8	20	$ab\star\star\star cde$	1	350
			$ab\star\star c\star de$	2	
			$ab\star\star cd\star e$	3	
			$ab\star\star cde\star$	4	
			$ab\star c\star\star de$	4	
			$ab\star c\star d\star e$	6	
			$ab\star c\star de\star$	8	
			$ab\star cd\star\star e$	9	
			$ab\star cd\star e\star$	12	
			$ab\star cde\star\star$	16	
			$abc\star\star\star de$	8	
			$abc\star\star d\star e$	12	
			$abc\star\star de\star$	16	
			$abc\star d\star\star e$	18	
			$abc\star d\star e\star$	24	
			$abc\star de\star\star$	32	
			$abcd\star\star\star e$	27	
			$abcd\star\star e\star$	36	
			$abcd\star e\star\star$	48	
			$abcde\star\star\star$	64	

Table 5.5: Example NNR Partitions for $n = 4$ and $n = 5$

In number theory, the number of compositions of n items into k parts is given by the binomial coefficient $\binom{n-1}{k-1}$. Each NNR partition is a composition of $l-1$ entries into $n-1$ segments of sizes $|s_2|, \cdots, |s_n|$, with $|s_1|$ always one. Therefore, the number of partitions is:

$$|\Gamma| = \binom{l-2}{n-2} = \frac{(l-2)!}{(n-2)!(l-n)!} \tag{5.9}$$

For the example of $n = 4$ and $l = 8$ in Table 5.5:

$$|\Gamma| = \binom{6}{2} = \frac{6!}{2! \times 4!} = \frac{720}{48} = 15 \tag{5.10}$$

Table 5.6 gives the number of NNR partitions $|\Gamma(n, l)|$ for $n \in [1, 9]$ and $l \in [1, 20]$.

$$|\Gamma| = \tfrac{(l-2)!}{(n-2)!(l-n)!} \qquad n > 1,\ l > 1,\ l \geq n$$

	n								
l	1	2	3	4	5	6	7	8	9
1	1								
2		1							
3		1	1						
4		1	2	1					
5		1	3	3	1				
6		1	4	6	4	1			
7		1	5	10	10	5	1		
8		1	6	15	20	15	6	1	
9		1	7	21	35	35	21	7	1
10		1	8	28	56	70	56	28	8
11		1	9	36	84	126	126	84	36
12		1	10	45	120	210	252	210	120
13		1	11	55	165	330	462	462	330
14		1	12	66	220	495	792	924	792
15		1	13	78	286	715	1287	1716	1716
16		1	14	91	347	1001	2002	3003	3432
17		1	15	105	404	1253	2772	4881	6414
18		1	16	120	560	1820	4368	6780	10466
19		1	17	136	680	2380	6188	12376	19448
20		1	18	153	816	3060	8568	18564	31824

Table 5.6: NNR Partitions $|\Gamma|$

For a particular partition γ the number of NNR sequences $|\gamma|$ can be determined from the partition sizes $|s_1|, |s_2|, \cdots, |s_n|$. The first segment s_1 is always a, due to the non-repeating characteristic of NNR sequences. The second segment s_2 begins with b, followed by an alternating sequence of a and b. The third segment s_3 begins with c, followed by a sequence of a, b and c. In general, the i^{th} segment begins with x_i followed by a choice of x_j where $j \in [1, i)$. Each subsequent entry in the segment can also be chosen from $i - 1$ elements, resulting in $(i-1)^{|s_i|-1}$ possible segment choices. Combined for all segments:

$$|\gamma| = 1^{|s_2|-1} \times \; 2^{|s_3|-1} \times \; 3^{|s_4|-1} \times \; \cdots \; \times \; (n-1)^{|s_n|-1} \tag{5.11}$$

As an example the 13th partition of $n = 4$ and $l = 8$ in Table 5.5 consists of segments $|s_1| = 1, |s_2| = 1, |s_3| = 3, |s_4| = 3$ (also denoted $abc{\star}{\star}d{\star}{\star}$). There are 36 NNR sequences for this partition, of the total NNR search space of 301:

$$|\gamma_{abc\star\star d\star\star}| = 1^0 \times 2^2 \times 3^2 = 1 \times 4 \times 9 = 36 \tag{5.12}$$

Our C++ approaches to computing the number of NNR partitions, and the size of each partition appear in Appendix B.5.2 and Appendix B.5.3, respectively.

5.4.6 Partition Based NNR Sequences

The previous section introduced the concept of an NNR partition $\gamma \in \Gamma$ which represents a distinct subset of the complete NNR search space. Given n and l the number of partitions $|\Gamma|$ and the number of NNR sequences in each partition $|\gamma|$ can be computed. This section gives a method in Algorithm 5.3 for generating the i^{th} NNR sequence in a particular partition γ where $i \in [0, |\gamma|)$.

Generating NNR sequences in this manner is more flexible than the recursive approach previously discussed in Algorithm 5.1. NNR sequences could be randomly sampled from the entire NNR search space, or sampled from a specific partition. It is also possible to narrow an exhaustive search to NNR sequences of partitions that are considered more likely to contain PESs.

Knowing a particular partition γ, and the number of possible NNR sequences in the partition $|\gamma|$, Algorithm 5.3 generates the i^{th} sequence in the partition. At the beginning of each segment the j^{th} element is appended to the sequence. Within each segment, elements are chosen from the $j - 1$ possible alternatives to the previous entry, and the index i is adjusted as these are appended to the sequence.

Our C++ implementation appears in Appendix B.5.4.

Algorithm 5.3 NNR Partition Sequence

```
Require: $\gamma \in \Gamma$ is the partition of interest
Require: $i \in [0, |\gamma|)$ is the index of the NNR sequence of interest
Require: $n$ is the number of elements (and partition segments)
Require: $l$ is the NNR sequence length
Require: $s_j$ is the segment, for $j \in [1, n]$
Require: $|s_1| = 1$
Require: $|s_j| \geq 1$ for $j \in [1, n]$
Require: $l = \sum_{j=1}^{n} |s_j|$
  sequence $\leftarrow \{ \}$
  for all segments $s_j$ with $j \in [1, n]$ do
    sequence $\leftarrow$ sequence $\cdot\ x_j$
    if $j > 1$ then
      for all $k \in [2, |s_i|]$ do
        $c \leftarrow i\ \%\ (j - 1)$                                [Integer modulus division]
        $i \leftarrow \frac{i}{j-1}$                                        [Integer division]
        $prev \leftarrow$ last element in sequence
        sequence $\leftarrow$ sequence $\cdot\ x_{(prev+1+c)\%j}$          [Integer modulus division]
```

5.4.7 A Lower Bound of $\Omega(n^2)$ for Shortest Length PESs

In computer science $O(f(n))$ denotes the order of an upper bound and $\Omega(f(n))$ denotes the order of a lower bound.[54] This section provides a proof of an $\Omega(n^2)$ lower bound for shortest length PESs based on the *Normalised No Repeat* (NNR) and partitioning concepts introduced in previous sections.

Consider the last segment s_n in the NNR partition: $s_1 \cdot s_2 \ldots s_n$. The segment s_n begins with the n^{th} element followed by a non-repeating sequence of all n elements. By definition, this segment is the only one containing the n^{th} element.

For an NNR sequence to embed permutations of n elements, the segment s_n must embed all permutations of $n - 1$ elements, corresponding to the permutations beginning with the n^{th} element. Therefore, the length of the segment s_n is at least the shortest length PES for $n - 1$, plus one. The length of the other segments must be at be at least one, resulting in:

$$f(1) = 1 \tag{5.13}$$

$$f(n) = f(n-1) + n \qquad \text{for } n > 1 \tag{5.14}$$

$$\therefore f(n) = f(n-2) + (n-1) + n \tag{5.15}$$

$$= 1 + \cdots + (n-2) + (n-1) + n \tag{5.16}$$

$$= n + (n-1) + (n-2) + \cdots + 1 \tag{5.17}$$

$$\therefore f(n) + f(n) = (n+1) + \cdots + (n+1) \qquad (5.16) + (5.17) \tag{5.18}$$

$$2f(n) = n(n+1) \tag{5.19}$$

$$f(n) = \frac{n^2+n}{2} \tag{5.20}$$

Giving a lower bound for PES length as:

$$|\mathrm{PES}_n| \geq (n \ + \ |\mathrm{PES}_{n-1}|) \tag{5.21}$$

$$|\mathrm{PES}_n| \geq \frac{n^2+n}{2} \tag{5.22}$$

This $\Omega(n^2)$ PES length lower bound excludes the possibility of asymptotic improvement to $O(n^2)$ PES length. Consequently the SCS CSG rendering of convex subtraction is firmly established as being $O(n^2)$ time.

We can also be certain that no better than a two times SCS CSG rendering speedup could be achieved using shorter length PESs than those of length $n^2 - 2n + 4$.

5.4.8 Known Shortest Length PESs

It is currently feasible on a desktop PC to exhaustively generate and test the NNR search space for $1 \leq n \leq 5$. Table 5.7 gives the complete set of shortest length PESs for $1 \leq n \leq 5$. It takes a 2Ghz Pentium 4 around 12 hours to test all NNR sequences for $n = 5,\ l = 19$. We estimate that searching all NNR sequences for $n = 6,\ l = 27$ would take around 100 years on the same machine. We have not found a PES of shorter length than those constructed using the methods of Adleman [14], Galbiati [35] or Savage [83], and no others have been reported.

We have also searched for shortest length kPESs, given in Table 5.8. As in the previous case, the size of the NNR space multiplied with the $n!$ embedded sequence test for each candidate substantially limits the size of n, k and l that can be handled by current technology. This experiment also confirmed Savage [83] sequences as being shortest length for those cases that could be manageably searched.

n	l	PES			$n^2 - 2n + 4$ for $n \geq 3$
1	1	*a*			
2	3	*aba*			
3	7	*abcabac*	*abcabca*	*abcacba*	7
		abcbabc	*abcbacb*	*abacaba*	
		abacbab			
4	12	*abcdabcadbac*	*abcdabcadbca*	*abcdabcadcba*	12
		abcdacbadbca	*abcdacbadcab*	*abcdacbadcba*	
		abcdbacbdabc	*abcdbacbdacb*	*abcdbacbdcab*	
5	19	*abcdeabcdaebcadbcea*	*abcdeabcdaebcadbeac*	*abcdeabcdaebcadbeca*	19
		abcdeabcdaebcadcbea	*abcdeabcdaebcadceba*	*abcdeabcdaebcadebac*	
		abcdeabcdaebcadebca	*abcdeabcdaebcadecba*	*abcdeabcdaecbadbcea*	
		abcdeabcdaecbadbeca	*abcdeabcdaecbadcbea*	*abcdeabcdaecbadceab*	
		abcdeabcdaecbadceba	*abcdeabcdaecbadebca*	*abcdeabcdaecbadecab*	
		abcdeabcdaecbadecba	*abcdeabdcaebdacbdea*	*abcdeabdcaebdacbead*	
		abcdeabdcaebdacbeda	*abcdeabdcaebdacdbea*	*abcdeabdcaebdacdeba*	
		abcdeabdcaebdacebad	*abcdeabdcaebdacebda*	*abcdeabdcaebdacedba*	
		abcdeabdcaedbacbdea	*abcdeabdcaedbacbeda*	*abcdeabdcaedbacdbea*	
		abcdeabdcaedbacdeab	*abcdeabdcaedbacdeba*	*abcdeabdcaedbacebda*	
		abcdeabdcaedbacedab	*abcdeabdcaedbacedba*	*abcdeacbdaebcadbcea*	
		abcdeacbdaebcadbeac	*abcdeacbdaebcadbeca*	*abcdeacbdaebcadcbea*	
		abcdeacbdaebcadceba	*abcdeacbdaebcadebac*	*abcdeacbdaebcadebca*	
		abcdeacbdaebcadecba	*abcdeacbdaecbadbcea*	*abcdeacbdaecbadbeca*	
		abcdeacbdaecbadcbea	*abcdeacbdaecbadceab*	*abcdeacbdaecbadceba*	
		abcdeacbdaecbadebca	*abcdeacbdaecbadecab*	*abcdeacbdaecbadecba*	
		abcdeacdbaecdabcdea	*abcdeacdbaecdabcead*	*abcdeacdbaecdabceda*	
		abcdeacdbaecdabdcea	*abcdeacdbaecdabdeca*	*abcdeacdbaecdabecad*	
		abcdeacdbaecdabecda	*abcdeacdbaecdabedca*	*abcdeacdbaedcabcdea*	
		abcdeacdbaedcabceda	*abcdeacdbaedcabdcea*	*abcdeacdbaedcabdeac*	
		abcdeacdbaedcabdeca	*abcdeacdbaedcabecda*	*abcdeacdbaedcabedac*	
		abcdeacdbaedcabedca	*abcdeadbcaebdacbdea*	*abcdeadbcaebdacbead*	
		abcdeadbcaebdacbeda	*abcdeadbcaebdacdbea*	*abcdeadbcaebdacdeba*	
		abcdeadbcaebdacebad	*abcdeadbcaebdacebda*	*abcdeadbcaebdacedba*	
		abcdeadbcaedbacbdea	*abcdeadbcaedbacbeda*	*abcdeadbcaedbacdbea*	
		abcdeadbcaedbacdeab	*abcdeadbcaedbacdeba*	*abcdeadbcaedbacebda*	
		abcdeadbcaedbacedab	*abcdeadbcaedbacedba*	*abcdeadcbaecdabcdea*	
		abcdeadcbaecdabcead	*abcdeadcbaecdabceda*	*abcdeadcbaecdabdcea*	
		abcdeadcbaecdabdeca	*abcdeadcbaecdabecad*	*abcdeadcbaecdabecda*	
		abcdeadcbaecdabedca	*abcdeadcbaedcabcdea*	*abcdeadcbaedcabceda*	
		abcdeadcbaedcabdcea	*abcdeadcbaedcabdeac*	*abcdeadcbaedcabdeca*	
		abcdeadcbaedcabecda	*abcdeadcbaedcabedac*	*abcdeadcbaedcabedca*	
		abcdebacdbeacbdaceb	*abcdebacdbeacbdaebc*	*abcdebacdbeacbdaecb*	
		abcdebacdbeacbdcaeb	*abcdebacdbeacbdceab*	*abcdebacdbeacbdeabc*	
		abcdebacdbeacbdeacb	*abcdebacdbeacbdecab*	*abcdebacdbecabdaceb*	
		abcdebacdbecabdaecb	*abcdebacdbecabdcaeb*	*abcdebacdbecabdceab*	
		abcdebacdbecabdceba	*abcdebacdbecabdeacb*	*abcdebacdbecabdecab*	
		abcdebacdbecabdecba	*abcdebadcbeadbcadeb*	*abcdebadcbeadbcaebd*	
		abcdebadcbeadbcaedb	*abcdebadcbeadbcdaeb*	*abcdebadcbeadbcdeab*	
		abcdebadcbeadbceabd	*abcdebadcbeadbceadb*	*abcdebadcbeadbcedab*	
		abcdebadcbedabcadeb	*abcdebadcbedabcaedb*	*abcdebadcbedabcdaeb*	
		abcdebadcbedabcdeab	*abcdebadcbedabcdeba*	*abcdebadcbedabceadb*	
		abcdebadcbedabcedab	*abcdebadcbedabcedba*		

Table 5.7: Known Shortest Length PESs in NNR Form

n	**k**	**l**	kPES	$kn-2k+4$ for $n \geq k \geq 3$
1	1	1	a	
2	1	2	ab	
2	2	3	aba	
3	1	3	abc	
3	2	5	$abcab$	
3	3	7	$abcabac$	7
4	1	4	$abcd$	
4	2	7	$abcdabc$	
4	3	10	$abcdabcabd$	10
4	4	12	$abcdabcadbac$	12
5	1	5	$abcde$	
5	2	9	$abcdeabcd$	
5	3	13	$abcdeabcdabce$	13
5	4	16	$abcdeabcdabecabd$	16
5	5	19	$abcdeabcdaebcadbcea$	19
6	1	6	$abcdef$	
6	2	11	$abcdefabcde$	
6	3	16	$abcdefabcdefabcd$	16
7	1	7	$abcdefg$	
7	2	13	$abcdefgabcdef$	

Table 5.8: Known Shortest Length kPESs in NNR Form

5.5 Discussion

PES methods due to Adleman [14], Galbiati [35] and Savage [83] result in sequences of length $n^2 - 2n + 4$ for PESs and $kn - 2k + 4$ for kPESs. Although these have not been proven to be shortest length, no shorter sequences are known. The $O(n^2)$ lower bound for PES length excludes the possibility of PES of significantly shorter length.

Utilisation of a generate and test strategy and formulation of the *Normalised No Repeat* (NNR) search space allow computerised exhaustive search for shortest length PESs. This is performed in substantially less time than would be otherwise necessary by making use of known characteristics of shortest length permutation embedding sequences. Further refinements and optimisations would be necessary to tackle increasingly large search spaces for $n > 5$.

Adleman [14], Galbiati [35] and Savage [83] sequences are therefore the best known general purpose subtraction sequences for CSG rendering. Whether these are shortest length, or search techniques can be refined and a counter-example located, remains an open problem.

5.5.1 Further Work

There are several aspects of the NNR search space that remain to be investigated. While these issues are of broader theoretical interest subsequent chapters focus on other aspects of SCS CSG rendering.

- The NNR search space could be further constrained, allowing exhaustive search for larger sized problems such as $n > 5$. For example the last partition s_n is known to have a certain minimum length.

- It may be possible to decrease the time required to test each candidate sequence. In our approach, each permutation is checked for the embedding property individually, in an arbitrary order. The test terminates with a fail condition as soon as any permutation is found not to be embedded. It may be possible to test multiple permutations or multiple candidates, as an alternative. Or, it may be possible to process the candidate sequence in a way that tests less likely embedded permutations earlier.

- The iterative NNR generation algorithm presented here results in lexicographically dissimilar adjacent sequences. It may be desirable to produce a more orderly sequence, and allow increased permutation embedding testing efficiently. Alternatively, it may be possible to order sequences in way that high quality candidates are clustered allowing the use of gradient search techniques.

- For $n \geq 4$, all known shortest length sequences are contained in the partition beginning with $abc \cdots x_n$ (Table 5.7). A search focused on this partition would be the most promising, and require less time than the entire NNR search space. Even though it is the largest partition, Table 5.5 suggests that it becomes a diminishing fraction as n and l increase.

- $n = 6$, $l = 27$ could be exhaustively searched with current technology using a distributed computation effort in the manner of Seti@Home [5] or Folding@Home [6]. Iterative generation of NNR sequences allows different machines to test different subsets of the search space.

Chapter 6

Overlap Graph

"We approached the task by starting with a simple scheme and adding commands and features that we felt would enhance the power of the machine. Gradually the processor became more complex. We were not disturbed by this because computer graphics, after all, *are* complex. Finally the display processor came to resemble a full-fledged computer with some special graphics features. And then a strange thing happened. We felt compelled to add to the processor a second, subsidiary processor, which, itself, began to grow in complexity. It was then that we discovered a disturbing truth. Designing a display processor can become a never-ending cyclical process. In fact, we found the process so frustrating that we have come to call it the *'wheel of reincarnation'*." [66]
— T. H. Myer and I. E. Sutherland

6.1 Introduction

As described in Chapter 4, the *Sequenced Convex Subtraction* (SCS) CSG rendering algorithm performs sequenced subtraction of convex shapes from the depth buffer. A discussion of *Permutation Embedding Sequences* (PESs) followed in Chapter 5, focusing on subtraction sequences of shortest length. In these previous chapters subtraction sequence encoding methods have been based on knowing the number of subtracted objects, and perhaps also the maximum depth complexity from a particular viewing direction.

This chapter approaches subtraction encoding by utilising additional information about the CSG tree — the object-space intersection (or overlap) between shapes. This overlap information is represented as a undirected graph, or a matrix. As a graph, each node corresponds to a shape and each edge corresponds to an overlap between a pair of shapes. As a matrix, each entry corresponds to an overlap between the shapes corresponding to a particular row and column.

The *overlap graph* is then processed into a subtraction sequence by examining the graph for particular properties. In the best case $O(n)$ subtraction sequences can be determined by taking this approach, although in the worst case view independent or view dependent subtraction sequences remain $O(n^2)$ in length. The length of subtraction sequences resulting from overlap graph processing are less than, or equal to, the length of the subtraction sequences discussed previously in Chapter 4 and Chapter 5.

6.1.1 Graph Theory Background

Here we introduce graph theory terminology and notation used in our work, based on that given in [27].

A *graph* $G = (V, E)$ is a set of *nodes* V and a set of *edges* E. Edges connect pairs of nodes in the graph. The *order* of a graph, denoted $|G|$ is the number of nodes in a graph. The number of edges in a graph is denoted $||G||$.

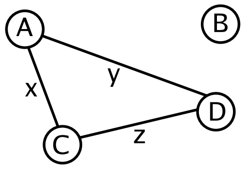
If $v \in V$, $e \in E$ and $v \in e$, then the node v is an *end* of edge e, edge e is *incident* on v, and e is an edge at node v. Two nodes x, y are *adjacent* if xy is an edge of G. Two edges are *adjacent* if they have an end in common. The *degree* of a node $d(v)$ is the number of edges incident at v. Nodes of degree zero are termed *isolated.* Nodes of degree one are called *leaves.*

A *path* is a graph $P = (V, E)$ linking two end nodes via intermediate nodes and edges. The degree of the end nodes in a path is 1 and the degree of the intermediate nodes is 2.

The *length* of a path is the number of edges. A *cycle* is a graph $C = (V, E)$ connecting the nodes of the cycle V into a loop along the edges E. The degree of the nodes in a cycle is 2. The *length* of a cycle is the number of nodes or edges. A graph containing no cycles is *acyclic.*

A graph is *connected* if every pair of nodes is connected by a path in G. Otherwise, the graph is *disconnected.* If U is a set of nodes, $G - U$ is obtained by deleting all the nodes in $U \cap V$ and their incident edges.

The disconnected graph in Figure 6.1 is composed of four nodes and three edges. A cycle is formed by nodes A, C and D with the other node B isolated.



$$
\begin{aligned}
&G = \{V, E\}\\
&V = \{A, B, C, D\}\\
&E = \{x, y, z\}\\
&|G| = 4\\
&||G|| = 3
\end{aligned}
$$

Figure 6.1: Graph Theory Notation

6.1.2 Overlap Graph Implementation

Our overlap graph is implemented in C++ as an adjacency-list using the C++ standard library `vector` and `map` templated containers. Use of a sparse representation is based on the assumption that there are generally few edges at each node. Sparse graphs are more likely to result in short subtraction sequences. We expect that real-world applications typically result in sparse graphs.

Specifically, the implementation uses a `std::vector` of `std::map` — one map for each node in the graph. Each map is a balanced tree of edges incident on a particular node. Each edge is stored at both end nodes. The class interface supports edge addition and node removal. Node degree and edge list queries are efficient.

Our own shape intersection testing routines have been used for overlap graph construction. In principle a collision detection library such as I-COLLIDE [25] could be used as an alternative. Optimisation of this aspect was not pursued in this investigation due to our focus on per-frame performance — overlap graph construction is a once off preprocessing step. If the CSG tree or shapes change over time the overlap graph also needs to be updated.

(a) $a \cup b$ (b) $a \cap b$ (c) $a - b$

Figure 6.2: Non-overlapping Intersection

6.2 Overlap Properties

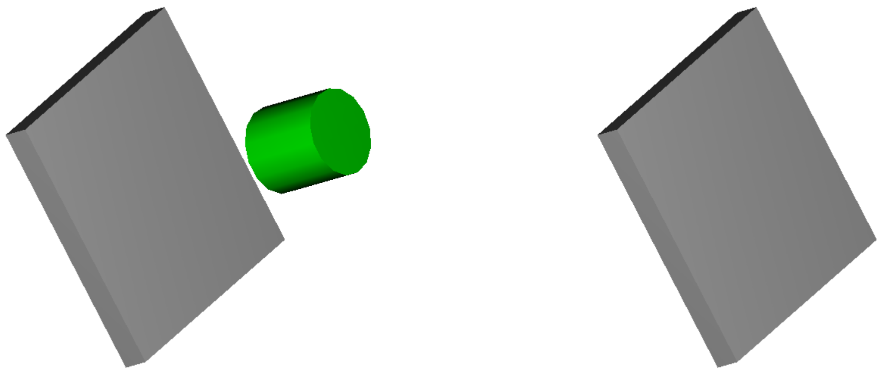
6.2.1 Intersected Objects

For a CSG product to be non-empty, all intersected objects in the product must overlap all other intersected objects in the product. If any pair of intersected objects in the product do not overlap then the whole product is empty and no rendering is required. Figure 6.2 illustrates (a) two non-overlapping objects, (b) the empty intersection and (c) the difference.

Algorithm 6.1 compares all pairs of intersected objects, returning `false` for any pair found not to be overlapping. *Axis-Aligned Bounding Boxes* (AABBs) are used in our implementation; other intersection testing methods could also be used. In this context false positives for intersection arising from the use of AABBs will affect rendering performance, but not the correctness of the end result. However, false negatives (two objects incorrectly classified as not intersecting) will lead to the product being ignored and left out of the rendered image. There is a tradeoff between the precision of intersection testing, the time spent intersection testing, and the time spent rendering the result. AABBs result in false positives but not false negatives. We found AABB intersection testing adequate for overlap graph purposes.

Algorithm 6.1 Intersected Objects Check

```
for all pairs of intersected objects: $I_i \in P$ and $I_j \in P$ do
    if $I_i$ and $I_j$ are not overlapping then
        return false                                        [CSG product is empty]
return true
```



(a) $x - a$ (b) Rendered Result

Figure 6.3: External Subtracted Object

6.2.2 External Subtracted Objects

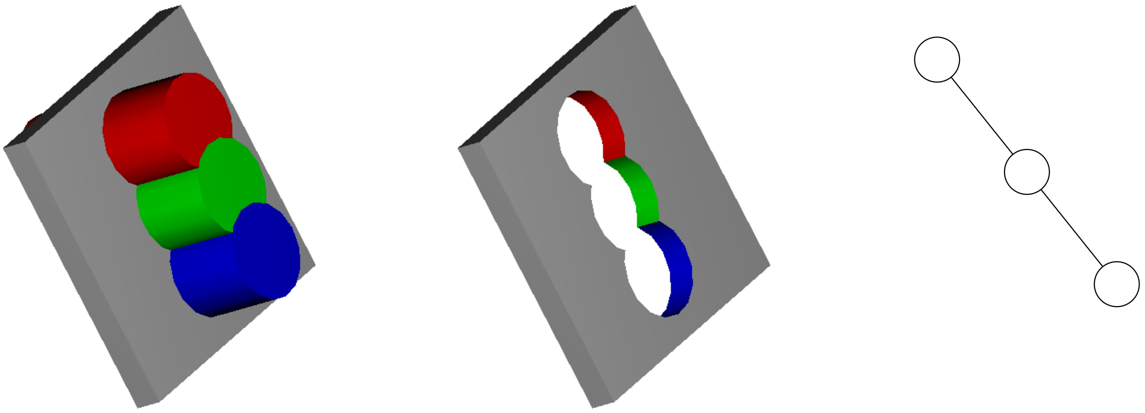
Surfaces of subtracted objects in the CSG product must be inside all intersected objects in the product to be visible. Subtracted objects in the product not overlapping all intersected objects in the product are external and can be omitted from the subtraction sequence.

In the example in Figure 6.3, a cylinder is subtracted from a rectangular block. The cylinder and block do not overlap, so the cylinder need not be subtracted. The resulting subtraction sequence for the set of external nodes is empty. This process is analogous to view-frustum culling — only subtracted objects overlapping particular areas of interest need to proceed to subsequent processing steps.

Algorithm 6.2 compares each subtracted object to all intersected objects. If the subtracted object is not overlapping every intersected object, the subtracted object is removed from the overlap graph.

Algorithm 6.2 External Subtracted Objects

```
for all subtracted objects: $S \in P$ do
  for all intersected objects: $I \in P$ do
    if $S$ and $I$ are not overlapping then
      remove $S$ from $G$                    [$S$ is not visible]
```



(a) $x - a - b - c$ (b) Rendered Result (c) Overlap Graph

Figure 6.4: Leaf Nodes

6.2.3 Leaf Node Trimming

Subtraction sequences include only the subtracted objects in the CSG product. The previous two tests make use of the spatial overlap of intersected objects in the product. Intersected objects are removed from the overlap graph at this stage leaving only subtracted nodes.

Leaf nodes are those having a degree of one (connected by one edge only) and represent subtracted objects that only overlap one other subtracted object in the overlap graph. Leaf node *trimming* can result in new leaf nodes, cyclic graphs, isolated nodes, or an empty graph. A set of removed leaf nodes is referred to as a *trim*. The overlap graph is trimmed until no further trimming is possible, resulting in a set of trims: $T_1, T_2, \cdots, T_n$. Repeated trimming results in cyclic graphs, isolated nodes, or an empty graph. An acyclic overlap graph is illustrated in Figure 6.4, with initial leaf nodes of a and c. The iterative leaf trimming algorithm is given in Algorithm 6.3.

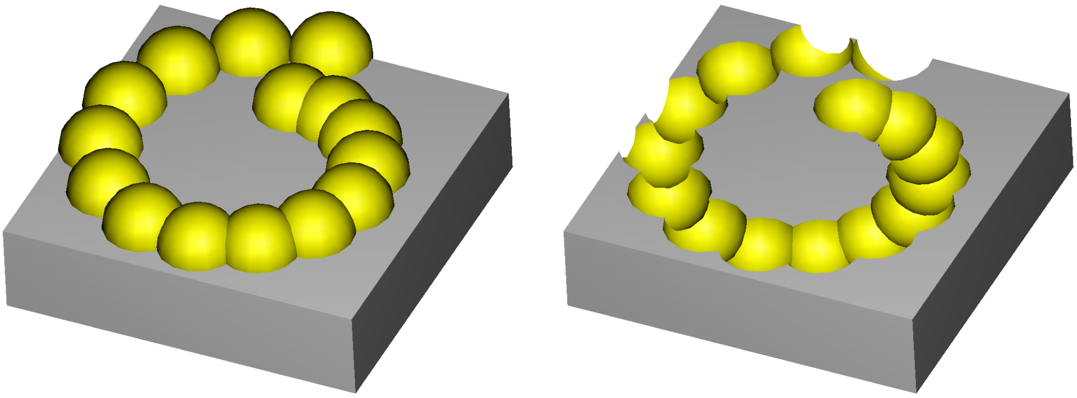
Algorithm 6.3 Leaf Node Trimming

```
$T_i$ is the trim in the $i^{\text{th}}$ pass
$i \leftarrow 1$
**while** leaf nodes exist **do**
  $T_i \leftarrow \textit{leaves}$
  remove $T_i$ from $G$
  $i \leftarrow i + 1$
```

Subtraction sequence encoding is based on the observation that necessary sequences of subtraction proceed from the outer trims towards the inner trims and then back out towards the outer trims. The trims are combined into the subtraction sequence $T_1 \cdot T_2 \cdot ... \cdot T_n \cdot S(G - T_1 - T_2 - ... - T_n) \cdot T_n \cdot ... \cdot T_2 \cdot T_1$ with $S(G - T_1 - T_2 - \cdots - T_n)$ being the subtraction sequence for the graph resulting from leaf trimming.

In the example in Figure 6.4, three cylinders are subtracted from a rectangular block. The two outer cylinders are trimmed as leaves in the first pass: $T_1 = a, c$. The combined subtraction sequence is: $acbac$. Each leaf node appears in the subtraction sequence twice, resulting in $O(n)$ length subtraction sequences for completely acyclic overlap graphs.

In the example in Figure 6.5, the fifteen subtracted spheres form an acyclic overlap graph. The first trim consists of the spheres at each end: $T_1 = a, o$. The second trim consists of the spheres second from each end: $T_2 = b, n$. Seven leaf trimming passes form a combined subtraction sequence of: $aobncm \cdots cmbnao$, an $O(n)$ length subtraction sequence. If this spiral was adjusted to form a loop the overlap graph would no longer be acyclic. In the next section we will describe how ring graphs can also be encoded into $O(n)$ length subtraction sequences.



(a) $x - a - b - \cdots - n - o$

(b) Rendered Result

Figure 6.5: Acyclic Overlap Graph Example

6.2.4 Ring Graphs

Left node trimming may result in an empty graph, or a collection of cyclic graphs and isolated nodes. Cyclic graphs in a ring formation can be identified and encoded as $O(n)$ subtraction sequences. This section discusses the procedure of overlap graph subtraction sequences for so-called "ring graphs". Each ring graph is identified and encoded individually.

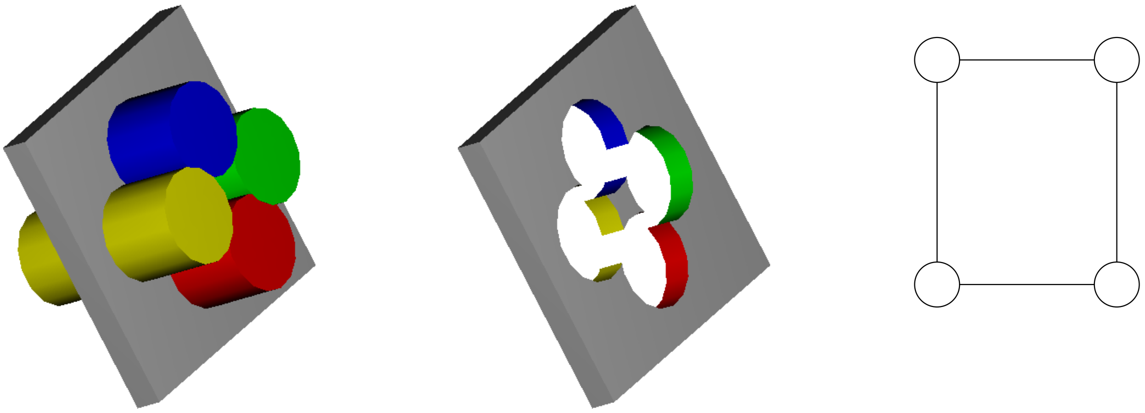
A ring is formed by a set of degree two nodes connected into a loop. Figure 6.6 illustrates four cylinders (a) subtracted from a rectangular block (b) forming an overlap graph ring (c).

Subtraction sequences for ring graphs need to embed all forward and reverse traversals of the ring. The sequence $R_1 \cdots R_n R_1 \cdots R_{n-1}$ embeds all forward traversals. The sequence $R_n \cdots R_1 R_n \cdots R_2$ embeds all reverse traversals. With the length of each of these sequences being $2n - 1$, the combined subtraction sequence length is $4n - 2$.

For the subtracted cylinders in Figure 6.6 the embedding of forward and reverse traversals into the sequence *abcdabcdcbadcb* is as follows:

	$R_1 \cdots R_n R_1 \cdots R_{n-1}$
forward	*abcdabc*
abcd	$abcd\star\star\star$
bcda	$\star bcda\star\star$
cdab	$\star\star cdab\star$
dabc	$\star\star\star dabc$

	$R_n \cdots R_1 R_n \cdots R_2$
reverse	*dcbadcb*
dcba	$dcba\star\star\star$
cbad	$\star cbad\star\star$
badc	$\star\star badc\star$
adcb	$\star\star\star adcb$



(a) $x - a - b - c - d$ (b) Rendered Result (c) Overlap Graph

Figure 6.6: Ring Graph

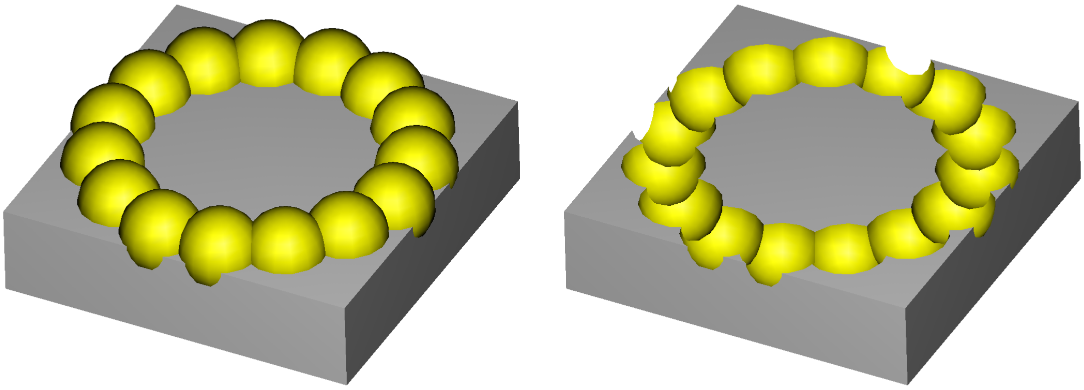
Algorithm 6.4 Ring Graph

```
Require: $R_1, \cdots, R_k$ are nodes forming a ring
  for all nodes of degree two: $i$ do
    $R_1 \leftarrow i$
    $R_2 \leftarrow$ overlapping node of $R_1$
    $k \leftarrow 2$
    while degree of $R_k$ is two do
      $R_{k+1} \leftarrow$ overlapping node of $R_k$ where $R_{k+1} \neq R_{k-1}$
      if $R_{k+1} = R_1$ then
        remove $R_1, \cdots, R_k$ from $G$                [Found a ring]
        return $R_1, \cdots, R_k$
      $k \leftarrow k + 1$
```

The method for finding an overlap ring graph is given in Algorithm 6.4. A depth-first search is used to visit nodes of degree two (node connected to two others) until the ring starting node is found. The algorithm is applied until there remains no further rings in the overlap graph.

In the example in Figure 6.7, the fifteen subtracted spheres form a ring overlap graph. The subtraction sequence is an $O(n)$ length subtraction sequence, compared to the $O(n^2)$ subtraction sequence that would be necessary otherwise.

The next section will discuss the problem of shortest length ring graph subtraction sequences, as a further performance improvement over the $4n - 2$ length sequences here.



(a) $x - a - b - \cdots - n - o$ (b) Rendered Result

Figure 6.7: Ring Overlap Graph Example

6.2.5 Shortest Length Ring Graph Sequences

The ring subtraction sequences described previously in Section 6.2.4 are $4n - 2$ in length. Shorter length sequences have been found using the search techniques discussed in Chapter 5. Shortest length ring subtraction sequences for $n \leq 6$ are given in Table 6.1. For $n > 6$ sequences shorter than $4n - 2$ have been found, but are not known to be shortest length. These are given in Table 6.2. A systematic way of generating shortest ring subtraction sequences without exhaustive searching remains unresolved.

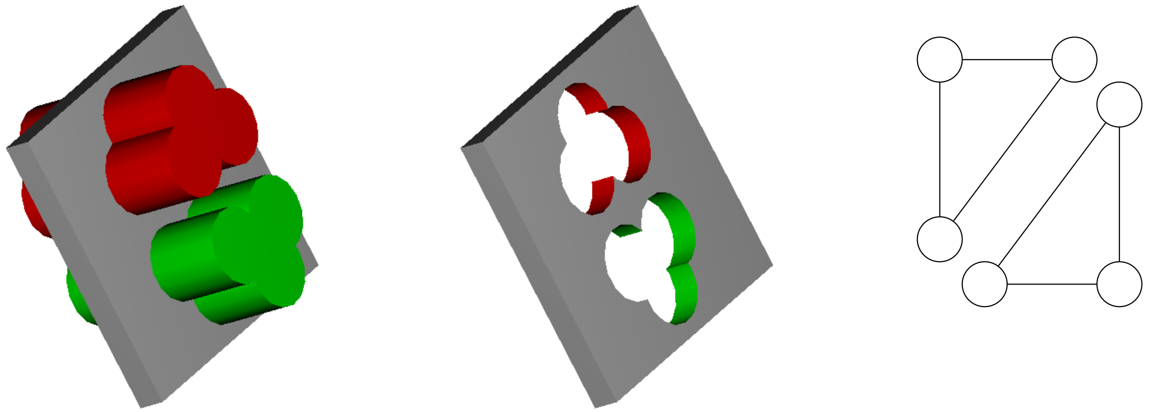
Our implementation makes use of a table of best-known ring subtraction sequences for $3 \leq n \leq 10$, or uses the general method in Section 6.2.4 for $n > 10$, resulting in $4n-2$ length sequences.

n	**l**	sequence	$4n - 2$
3	7	$abcabca$	10
		$abcbabc$	
		$abcabac$	
		$abcacba$	
		$abcbacb$	
		$abacbab$	
		$abacaba$	
4	10	$abcdcbadbc$	14
		$abcdcbabdc$	
		$abcdcabadc$	
		$abcdacbadc$	
5	14	$abcdedcbaedbcd$	18
6	18	$abcbdaefdcebabecdf$	22

Table 6.1: Shortest Length Ring Sequences

n	**l**	sequence	$4n - 2$
7	22	$abcdbaefgfedabcbadfegf$	26
8	26	$abcdedcbafghagdfbcedcbfahg$	30
9	30	$abcdefgedcbhiaibhcgdfedcbfgaih$	34
10	35	$abcdefghgfiedcbajaighfedcbacdjefhgi$	38

Table 6.2: Shortest Known Ring Sequences



(a) $x - a - b - c - d - e - f$ (b) Rendered Result (c) Overlap Graph

Figure 6.8: Disconnected Graphs

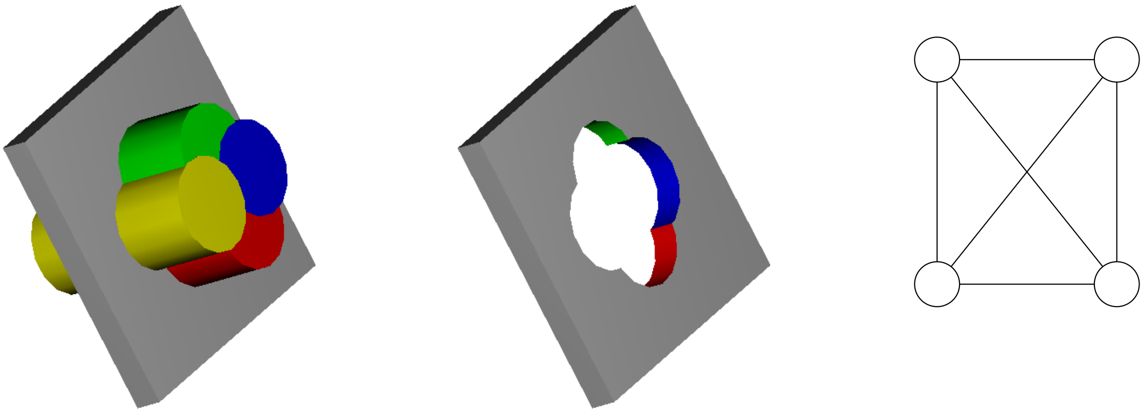
6.2.6 Disconnected Graphs

An overlap graph may be disconnected, that is, composed of separate connected components. Each connected component is treated individually when converted to a subtraction sequence. There is no need to embed sequences between disconnected portions of the overlap graph. This results in shorter sequences as the sum of the squares is less than the square of the sum, in general.

Individual connected components can be identified by starting from any node and traversing all the known edges until no unvisited nodes can be found. This can be implemented as either a depth-first or breadth-first traversal.

Each component can be encoded separately and combined into a concatenated subtraction sequence without concern for the order — as separate connected components in the overlap graph have no interdependence. Figure 6.8 illustrates six subtracted objects forming a disconnected graph with two connected components.

Isolated overlap graph nodes (those with a degree of zero) can be treated as trivial connected components. The subtraction sequence for an isolated node is simply the node itself — the object need only be subtracted once to ensure the correctly rendered result.



(a) $x - a - b - c - d$ (b) Rendered Result (c) Overlap Graph

Figure 6.9: Cyclic Graph

6.2.7 Cyclic Graphs

Cyclic graphs that are not rings are encoded as either $O(n^2)$ view-independent or $O(kn)$ view-dependent 'image-space' subtraction sequences as discussed in Chapter 4 and 5.

In the example in Figure 6.9, the four subtracted cylinders all overlap each other. None of the nodes are external to the block or are leaf nodes. Also, the cylinders do not form a ring. In this case either a $O(n^2)$ view-independent or a $O(kn)$ view-dependent subtraction sequence must be used.

6.3 Overlap Graph Subtraction Sequences

6.3.1 Sequence Encoding Algorithm

The overlap graph methods discussed in previous sections are combined in Algorithm 6.5 to produce a subtraction sequence S for a given CSG product P with an overlap graph G. Apart from the $O(n^2)$ cyclic graph subtraction sequences, the resulting subtraction sequences are $O(n)$.

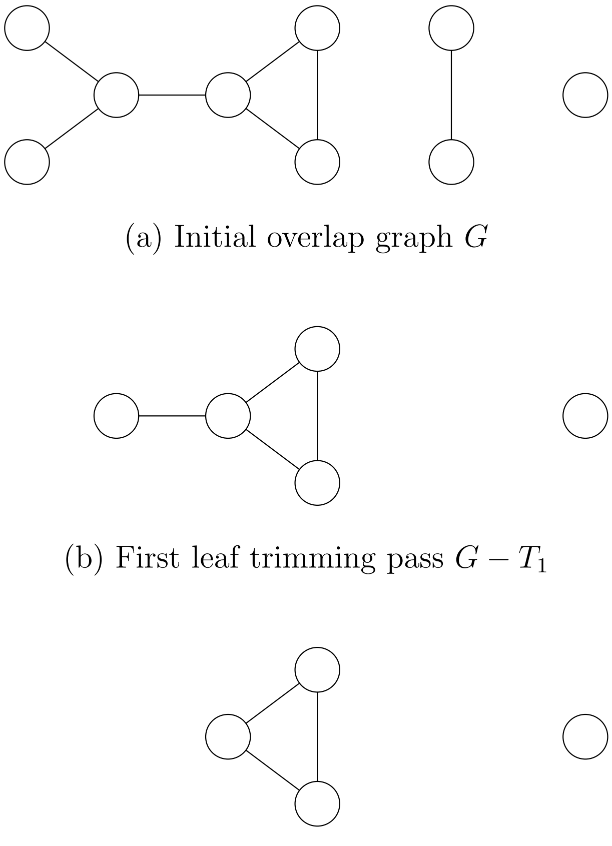
Algorithm 6.5 Overlap Graph Subtraction Sequence Encoding

```
Require: $P$ is the CSG product
Require: $G$ is the overlap graph
Require: $n_{trim}$ is the number of leaf node trims
Require: $n_{ring}$ is the number of ring graphs
Require: $n_{cyclic}$ is the number of cyclic graphs
Require: $S_{ring}(R_i)$ is the subtraction sequence for the $i^{\text{th}}$ ring
Require: $S_{cyclic}(C_i)$ is the subtraction sequence for the $i^{\text{th}}$ cyclic graph
Require: $L$ is the set of isolated nodes
  [Empty intersection — Algorithm 6.1]
  for all pairs of intersected objects: $I_i \in P$ and $I_j \in P$ do
    if $I_i$ and $I_j$ are not overlapping then
      return empty sequence
  [External subtracted objects — Algorithm 6.2]
  for all subtracted objects $S \in P$ do
    for all intersected objects $I \in P$ do
      if $S$ and $I$ are not overlapping then
        remove $S$ from $G$
  [Leaf node trimming — Algorithm 6.3]
  for all leaf trims $T_i \in G$ where $i \in [1, n_{trim}]$ do
    remove $T_i$ from $G$
  [Ring graphs — Algorithm 6.4]
  for all ring sub-graphs $R_i \in G$ where $i \in [1, n_{ring}]$ do
    remove $R_i$ from $G$
  [Cyclic connected graphs]
  for all connected cyclic graphs $C_i \in G$ where $i \in [1, n_{cyclic}]$ do
    remove $C_i$ from $G$
  [Combined subtraction sequence $S_P$ for product $P$]
  $S_P \leftarrow T_1 \cdot T_2 \cdot ... \cdot T_{n_{trim}}$                    [Append leaf nodes from outer to inner]
  $S_P \leftarrow S_P \cdot S_{ring}(R_1) \cdot S_{ring}(R_2) \cdot ... \cdot S_{ring}(R_{n_{ring}})$        [Append ring graph sequences]
  $S_P \leftarrow S_P \cdot S_{cyclic}(C_1) \cdot S_{cyclic}(C_2) \cdot ... \cdot S_{cyclic}(C_{n_{cyclic}})$  [Append cyclic graph sequences]
  $S_P \leftarrow S_P \cdot L$                                             [Append isolated nodes]
  $S_P \leftarrow S_P \cdot T_{n_{trim}} \cdot ... \cdot T_2 \cdot T_1$                  [Append leaf nodes from inner to outer]
  return $S_P$
```

6.3.2 Example

Subtraction sequence encoding using an overlap graph is illustrated in Figure 6.10. Nine subtracted objects are tested for mutual intersection and form the overlap graph in Figure 6.10(a). Intersected nodes are not shown — empty intersection and external subtracted object tests have already been applied. The result of the first leaf trimming pass is given in Figure 6.10(b). Four leaf nodes are removed in total, which form the prefix and suffix of the combined subtraction sequence. The second trimming pass removes an additional leaf node as illustrated in Figure 6.10(c). Now that no further leaves exist in the overlap graph, cyclic connected graphs are considered. The ring graph of size three is encoded as the first part of the inner subtraction sequence. Finally, the remaining (isolated) node is added to the inner subtraction sequence. In this case the length of the combined subtraction sequence is $O(n)$ and no depth complexity sampling for view-dependent sequence encoding is necessary.

The combined sequence is formed by surrounding the encoded cyclic graph and isolated node with the leaf trims: $T_1 \cdot T_2 \cdot S_{ring}(G_{ring}) \cdot G_{isolated} \cdot T_2 \cdot T_1$



(a) Initial overlap graph G

(b) First leaf trimming pass $G - T_1$

(c) Second leaf trimming pass $G - T_1 - T_2$

Figure 6.10: Overlap Graph Sequence Encoding Example

6.3.3 Discussion

This chapter introduced the concept of overlap graph processing for the purpose of subtraction sequence construction. Traversal of the undirected overlap graph facilitates detection of certain nodes that can be omitted from the general-purpose $O(n^2)$ subtraction sequence. A significant benefit of the approach is the $O(n)$ subtraction sequences possible for entirely acyclic overlap graphs. Application of overlap graph subtraction sequences to CSG rendering is discussed further in Chapter 7.

Chapter 7

Experimental Results

Doubt is not a pleasant condition, but certainty is absurd.

— Voltaire

7.1 Introduction

This chapter examines empirical performance aspects of graphics hardware based CSG rendering algorithms, with particular focus on our new *Sequenced Convex Subtraction* (SCS) algorithm. The previously reported algorithms discussed in Chapter 3 are compared to the SCS approach as described in Chapter 4, 5 and 6 utilising shortest length permutation embedding sequences and overlap graph techniques. The broader aims of this chapter are to explore the real-world motivation, rationale and benefits of the SCS approach to CSG rendering.

The performance of CSG rendering is sensitive to various aspects of the platform, including CPU, GPU, memory, bandwidth and graphics hardware and drivers. Section 7.2 focuses on graphics hardware performance characteristics. Of particular interest to CSG rendering is the performance of depth buffer and stencil buffer copying, within the frame buffer and between the frame buffer and main system memory. Traditionally, computer algorithms tend to be either CPU, memory or disk intensive. Graphics applications can also be graphics hardware limited, which is what we find to be the case for CSG rendering.

A uniform platform was used in Section 7.3 for the purpose of comparing CSG rendering performance for the Goldfeather [36, 37], Layered Goldfeather [87], Improved Layered Goldfeather [29] and SCS [88, 89, 90] algorithms. Three generations of Nvidia OpenGL graphics cards on a reference PC platform were used to investigate algorithm performance for a set of benchmark CSG models.

Section 7.4 focuses on performance characteristics of the SCS algorithm such as convex intersection, convex subtraction and subtraction sequences. Linear-time convex intersection is confirmed experimentally. Overlap graph subtraction sequences are shown to be advantageous in certain circumstances. The cost of overlap graph processing is examined as a proportion of total rendering time. The performance of concave versus convex representations is also examined.

A discussion and conclusion of our experimental results follows in Section 7.5.

7.2 OpenGL Buffer Copying

The performance of image-space hardware-based CSG rendering algorithms is limited by the capability and performance characteristics of 3D graphics hardware. One important characteristic is the buffer copying rate — the number of pixels that can be copied per unit time within the frame buffer, or copied between the frame buffer and system memory.

The Goldfeather algorithm makes particularly heavy use of buffer copying, due to the approach of clipping primitives or layers in one buffer and then merging those results in a second depth buffer. The SCS algorithm also uses this approach for merging CSG products in a second depth buffer.

In analysing the performance of various CSG rendering algorithms, it is therefore important to examine the buffer copying ability of a particular platform. In this experiment three different Nvidia-based graphics cards are benchmarked for three distinct buffer copying tasks:

- `glCopyPixels`
  Copy colour, stencil or depth pixels within the frame buffer.
- `glReadPixels`
  Read colour, stencil or depth pixels into system memory.
- `glDrawPixels`
  Draw colour, stencil or depth pixels from system memory into the frame buffer.

The three graphics cards tested:

- ASUS V8200 Deluxe GeForce3, 64MB of memory
- Leadtek A250^{LE} GeForce4 Ti 4200, 64MB of memory
- Leadtek A310 GeForceFX 5600, 128MB of memory

The machine used for testing was a 2.0GHz Pentium 4 with 512MB RAM running RedHat Linux 9 with version 44.96 of the Nvidia drivers at an X11 desktop resolution of 1280x1024. Each value in Tables 7.1, 7.2 and 7.3 is the average taken over approximately 30 seconds, sufficient time to average out the variability observed in buffer copying time.[1] The variation in buffer copying performance is not examined in this work.

[1] We find that the size of the OpenGL window does not tend to affect the copy rate.

Platform	glCopyPixels (10^6 pixel/s)	OpenGL Format	glReadPixels (10^6 pixel/s)	glDrawPixels (10^6 pixel/s)
GeForce3	320	RGBA GLbyte	43	10
		RGBA GLubyte	43	70
		BGRA GLbyte	43	0.8
		BGRA GLubyte	43	70
GeForce4 Ti 4200	470	RGBA GLbyte	43	10
		RGBA GLubyte	43	93
		BGRA GLbyte	43	0.8
		BGRA GLubyte	43	93
GeForceFX 5600	480	RGBA GLbyte	44	10
		RGBA GLubyte	44	71
		BGRA GLbyte	44	0.8
		BGRA GLubyte	44	71

Table 7.1: OpenGL Colour Buffer Copying Performance

The results in Table 7.1 indicate that the performance of glCopyPixels for colour data ranges from 320 to 480 million pixel/s. glReadPixels is approximately an order of magnitude slower — around 43 million pixel/s. The performance of glDrawPixels ranges from 0.8 to 93 million pixel/s, depending on the format, signed RGBA and signed BGRA are considerably slower than unsigned RGBA or unsigned BGRA.

Across the three generations of graphics hardware there is some performance improvement in glCopyPixels, but similarity for glReadPixels and glDrawPixels. This may be due to architectural and driver similarities — or that the PCI/AGP bus is the ultimate bottleneck for glReadPixels and glDrawPixels. Compared to the theoretical AGP 4X capacity of 8 Gb/s, 93 million pixel/s corresponds to 3 Gb/s. To give an idea, a 32 bit/pixel 1024x768 image at 100 Hz would require 79 million pixel/s — bandwidth of 2.5Gb/s. So it does not appear that AGP bus bandwidth is the limiting factor.

OpenGL stencil buffer copying performance results are given in Table 7.2. glCopyPixels for stencil data ranges from 320 to 420 million pixel/s. glReadPixels is around 43 million pixel/s. glDrawPixels ranges from 110 to 130 million pixel/s.

Platform	glCopyPixels (10^6 pixel/s)	OpenGL Format	glReadPixels (10^6 pixel/s)	glDrawPixels (10^6 pixel/s)
GeForce3	320	GLbyte	44	110
		GLubyte	44	110
GeForce4 Ti 4200	420	GLbyte	43	130
		GLubyte	43	130
GeForceFX 5600	320	GLbyte	44	115
		GLubyte	44	115

Table 7.2: OpenGL Stencil Buffer Copying Performance

Interestingly, glCopyPixels of 8-bit stencil data is copied at a fairly similar rate to 32-bit colour data (Table 7.1). Stencil glReadPixels performance is close to that of colour. Stencil glDrawPixels performance is somewhat better than colour — even though 8-bit stencil data is one quarter of the size of 32-bit colour data. Only minor differences are observed between the three cards tested.

The results for depth buffer copying are given in Table 7.3. They indicate that the performance of glCopyPixels for depth data is 0.50 million pixel/s, compared to 320–480 million pixel/s for colour or stencil data. glReadPixels is comparable to colour or stencil data at 43 million pixel/s, except for the GLfloat format. glDrawPixels is also comparable to colour or stencil for unsigned integer formats at 100–115 million pixel/s. There is little variation between the three generations of cards tested.

The performance of depth data copying within the frame buffer has significant implications for CSG rendering. For depth data it is far more efficient to copy between the frame buffer and main memory, than to copy within the frame buffer. The relatively good performance of colour and stencil copying within the frame buffer suggest that glCopyPixels depth performance could be dramatically improved in future hardware or driver releases.

The OpenGL architecture specifies that glCopyPixels rasterises depth data, and all of the fragment testing stages are applied — which could account for the relative slowness of copying depth data. However, glDrawPixels also rasterises depth data as it is copied,

Platform	glCopyPixels (10^6 pixel/s)	OpenGL Format	glReadPixels (10^6 pixel/s)	glDrawPixels (10^6 pixel/s)
GeForce3	0.50	GLshort	43	0.5
		GLushort	43	100
		GLint	43	0.5
		GLuint	43	100
		GLfloat	26	20
GeForce4 Ti 4200	0.50	GLshort	43	0.5
		GLushort	43	110
		GLint	43	0.5
		GLuint	43	115
		GLfloat	18	25
GeForceFX 5600	0.50	GLshort	44	0.5
		GLushort	44	110
		GLint	44	0.5
		GLuint	44	110
		GLfloat	18	12

Table 7.3: OpenGL Depth Buffer Copying Performance

applying the same fragment processing stages, so that is not a complete explanation.

Summarising these results:

- While colour or stencil data can be copied within the frame buffer at 320 to 480 million pixel/s, depth copying is considerably slower at 0.5 million pixel/s — for reasons unclear.
- The emulation of multiple depth buffers for CSG rendering on these graphics cards is most efficiently implemented using `glReadPixels` and `glDrawPixels`, rather than `glCopyPixels`.
- The performance of moving colour and depth data between the frame buffer and main memory is particularly sensitive to the format. For example, depth `glDrawPixels` throughput varies from 0.5 million pixel/s for `GLint` to 110 million pixel/s for `GLuint`.

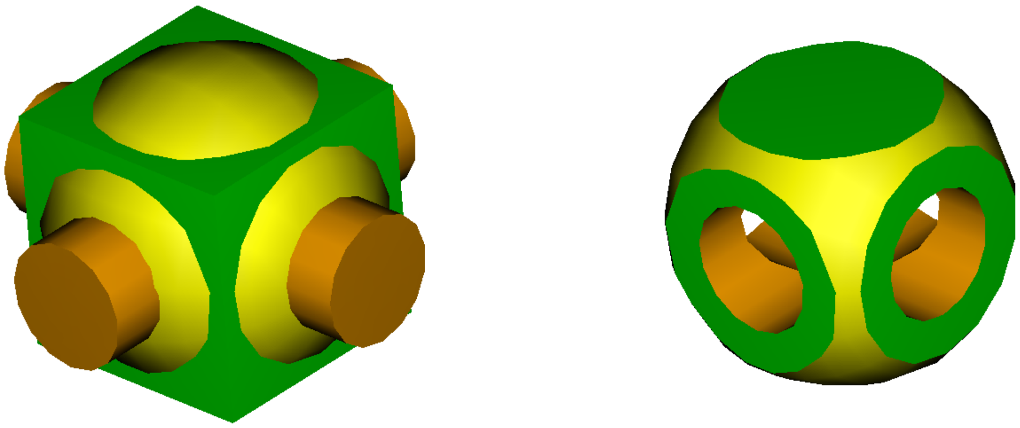
The results in following sections have all used depth buffer copying between the frame buffer and main memory to emulate multiple depth buffers in OpenGL. The results here reflect the hardware we used in our work, but may not reflect graphics hardware from other vendors or later products.

7.3 CSG Rendering Performance

This section examines the rendering performance of the Goldfeather [37], Layered Goldfeather [87], Improved Layered Goldfeather [29] and SCS [88, 89, 90] algorithms.

These experiments used the same hardware and configuration as detailed in Section 7.2. The time per frame appearing in each table is averaged over one thousand frames, each frame from a random viewing direction. The time per frame is also averaged across the three graphics cards, and listed as average frequency, time and relative time. The relative average time per frame summarises the performance of each algorithm for each CSG model. A relative time of 10.0 is ten times slower than the fastest measured algorithm for that model.

The CSG models in this section are composed of convex objects including spheres, cylinders and boxes. The Goldfeather algorithms can also handle concave geometry, but the SCS algorithm cannot. The issue of convex and concave geometry is specifically addressed in Section 7.4.5.



(a) Shapes

(b) $a \cap b - c - d$

Figure 7.1: CSG Widget Model

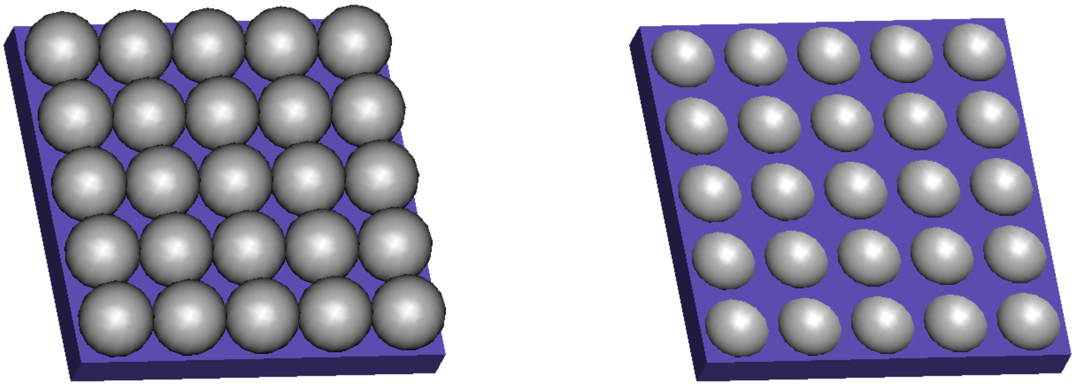
7.3.1 Widget

The widget model is a single CSG product composed of the intersection of a sphere and a box with two subtracted cylinders, as illustrated in Figure 7.1. This arrangement is representative of a typical machining operation. The depth complexity is the same from every viewing direction due to all of the objects overlapping each other. Timing results are given in Table 7.4.

The Layered Goldfeather algorithm is slower than the original Goldfeather algorithm since all the objects overlap and there is no advantage in clipping layers, rather than individual objects. The Improved Layered Goldfeather algorithm is the fastest Goldfeather variant, but is nine times slower than the SCS algorithm. The SCS algorithm is particularly advantageous in this case since no buffer copying is required for rendering a single CSG product.

Rendering Algorithm	time (s/frame)			average		relative
	GF3	GF4	FX	(s/frame)	(Hz)	time
Goldfeather	0.414	0.386	0.419	0.407	2.5	16.9
Layered Goldfeather	0.495	0.376	0.538	0.470	2.1	19.6
Imp. Layered Goldfeather	0.193	0.178	0.276	0.216	4.6	9.0
SCS	0.017	0.015	0.039	0.024	42.3	1.0

Table 7.4: CSG Widget Rendering Performance



(a) Shapes (b) $x - a - b - c - \cdots$

Figure 7.2: CSG Grid Model

7.3.2 Grid

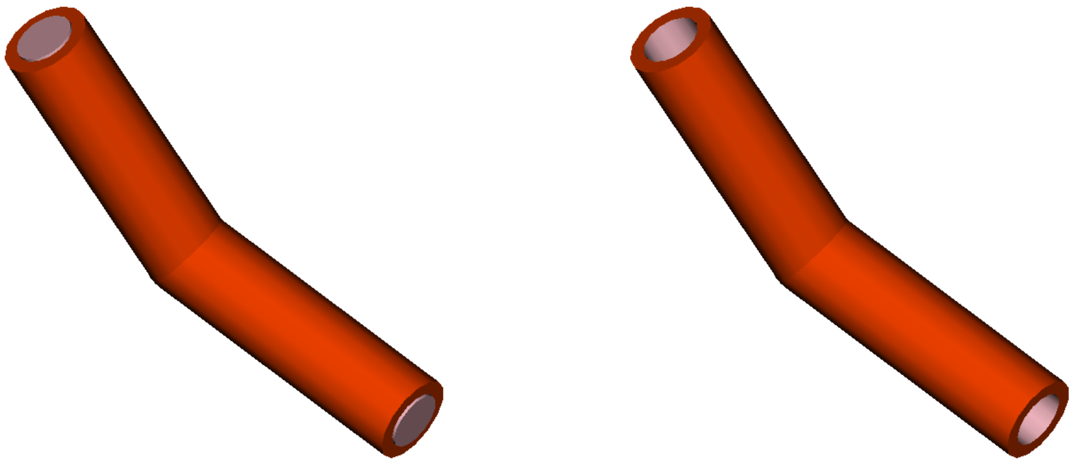
The grid model is a single CSG product composed of 25 non-overlapping spheres subtracted from a box, as illustrated in Figure 7.2. The depth complexity of the subtracted spheres ranges between one and ten, depending on the viewing direction. Timing results are given in Table 7.5.

The Goldfeather algorithm is particularly slow in this case — as each sphere is clipped against all of the other spheres even though there is no overlap between them. The Layered Goldfeather and Improved Layered Goldfeather algorithms provide a performance advantage with an average of 3.5 layers being substantially less than the 25 primitives.

The SCS algorithm is particularly advantageous in this case since no buffer copying is required. SCS is four times faster on average than the Improved Layered Goldfeather algorithm for this model.

Rendering Algorithm	time (s/frame)			average		relative
	GF3	GF4	FX	(s/frame)	(Hz)	time
Goldfeather	3.426	3.164	3.187	3.259	0.3	42.3
Layered Goldfeather	0.565	0.421	0.551	0.512	2.0	6.6
Imp. Layered Goldfeather	0.315	0.279	0.446	0.347	2.9	4.5
SCS	0.069	0.060	0.102	0.077	13.0	1.0

Table 7.5: CSG Grid Rendering Performance



(a) Shapes

(b) $(a \cup b \cup c) - (d \cup e \cup f)$

Figure 7.3: CSG Pipe Model

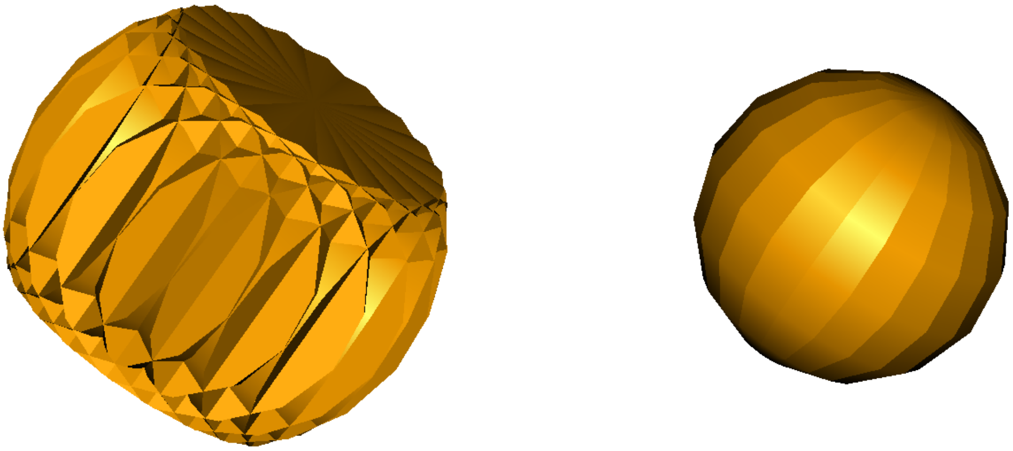
7.3.3 Pipe

The pipe model is composed of cylinders with a spherical joint, as illustrated in Figure 7.3. The inner cylinders and sphere (not shown) are subtracted from the outer cylinders and sphere forming a hollowed out tube. The CSG tree in normalised form is: $(a - d - e - f) \cup (b - d - e - f) \cup (c - d - e - f)$. Timing results are given in Table 7.6.

With three CSG products, the SCS algorithm makes use of buffer copying, but is twice as fast as the Improved Layered Goldfeather algorithm. The Layered Goldfeather algorithm is the slowest algorithm, nearly ten times slower than SCS.

Rendering Algorithm	time (s/frame)			average		relative
	GF3	GF4	FX	(s/frame)	(Hz)	time
Goldfeather	1.444	1.364	1.247	1.351	0.7	5.6
Layered Goldfeather	2.676	1.964	2.374	2.338	0.4	9.7
Imp. Layered Goldfeather	0.479	0.448	0.607	0.511	2.0	2.1
SCS	0.278	0.202	0.245	0.242	4.1	1.0

Table 7.6: CSG Pipe Rendering Performance



(a) Shapes

(b) $a \cap b \cap c \cap \cdots$

Figure 7.4: CSG Intersected Cylinders Model

7.3.4 Intersected Cylinders

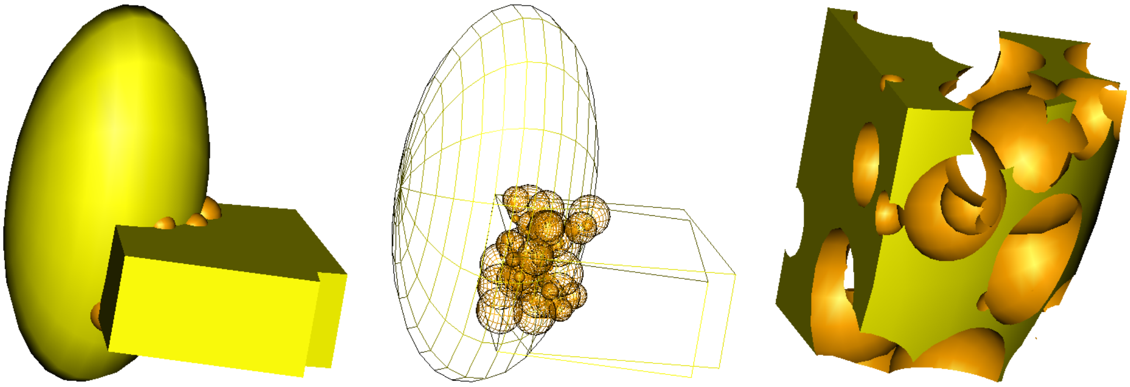
The cylinders model is a single CSG product composed of ten intersected cylinders, as illustrated in Figure 7.4. With all the cylinders intersecting, the depth complexity is ten. Timing results are given in Table 7.7.

The SCS algorithm is substantially faster than the Goldfeather algorithms due to the efficiency of intersection of convex shapes as implemented in SCS. No buffer copying or subtraction sequences are necessary in this special case. On the three graphics cards we tested, SCS is 29 times faster than the Improved Layered Goldfeather algorithm on average.

The linear-time characteristic of SCS intersection is confirmed later in Section 7.4.1 by measuring the rendering time for a varying number of intersecting cylinders.

Rendering Algorithm	time (s/frame)			average		relative
	GF3	GF4	FX	(s/frame)	(Hz)	time
Goldfeather	1.297	1.188	1.309	1.265	0.8	39.5
Layered Goldfeather	1.475	1.391	1.773	1.546	0.6	48.3
Imp. Layered Goldfeather	0.710	0.631	1.448	0.930	1.1	29.0
SCS	0.028	0.020	0.048	0.032	31.3	1.0

Table 7.7: CSG Intersected Cylinders Rendering Performance



(a) Shapes

(b) Wireframe

(c) $(a \cap b \cap c) - x - y - \cdots$

Figure 7.5: CSG Swiss Cheese Model

7.3.5 Swiss Cheese

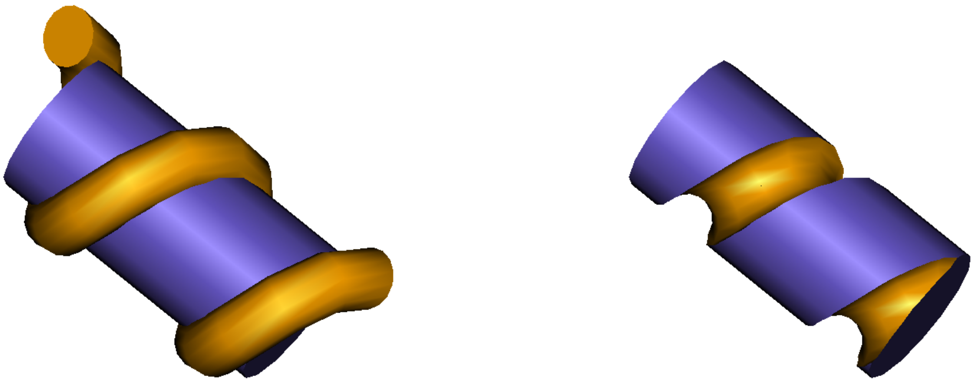
The Swiss cheese model consists of fifty spheres subtracted from the intersection of an ellipsoid and two boxes, as illustrated in Figure 7.5. The subtracted spheres resembling the holes in Swiss cheese are randomly positioned and scaled. The depth complexity of the spheres varies from eight to 14, with an average of 10.6. Timing results are given in Table 7.8.

The SCS algorithm is the most efficient on all three graphics cards, being over three times faster than the Improved Layered Goldfeather algorithm on average. The advantage of the layered Goldfeather approach, relative to the original algorithm is illustrated in this situation with the depth complexity being considerably less than the number of objects.

The $O(n^2)$ rendering passes of the Goldfeather algorithm could be improved. Weigand [97] suggests optimisations including object-space bounding box tests to reduce the number of clipping passes for each layer in the frame buffer. Although we would expect this to improve performance it does not resolve the common bottleneck — the $O(n)$ depth buffer copying operations.

Rendering Algorithm	time (s/frame)			average		relative
	GF3	GF4	FX	(s/frame)	(Hz)	time
Goldfeather	7.895	7.204	7.798	7.632	0.1	19.8
Layered Goldfeather	2.051	1.510	2.101	1.887	0.5	4.9
Imp. Layered Goldfeather	1.279	1.115	1.932	1.442	0.7	3.7
SCS	0.313	0.259	0.587	0.386	2.6	1.0

Table 7.8: CSG Swiss Cheese Rendering Performance



(a) Shapes

(b) $x - a - b - c - \cdots$

Figure 7.6: CSG Tool Model

7.3.6 Tool

The tool model is a single CSG product composed of twenty convex segments of a concave helix subtracted from a cylinder, as illustrated in Figure 7.6. The depth complexity of the subtracted segments ranges between three and seven, with an average of 4.8. Timing results are given in Table 7.9.

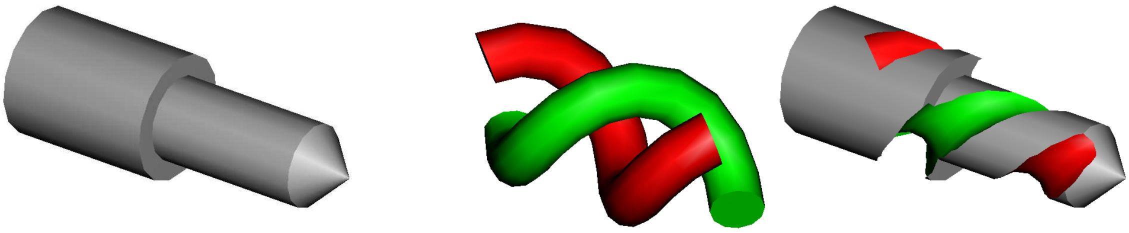
While the Goldfeather algorithm is the slowest, the Layered Goldfeather algorithms benefit from utilising the relatively low depth complexity.

The SCS algorithm is the fastest, being over six times faster than the Improved Layered Goldfeather algorithm on average. The single CSG product requires no buffer copying and low depth complexity is beneficial for minimising subtraction sequence length.

Overlap graph leaf trimming is particularly advantageous for this model with the subtracted segments arranged in a strand from one end to the other. This results in a more linear-time performance characteristic (with respect to the number of helix segments) than $O(n^2)$ in the general case.

Rendering Algorithm	time (s/frame)			average		relative
	GF3	GF4	FX	(s/frame)	(Hz)	time
Goldfeather	2.703	2.508	2.563	2.591	0.4	27.3
Layered Goldfeather	1.431	1.045	1.422	1.299	0.8	13.7
Imp. Layered Goldfeather	0.566	0.505	0.872	0.648	1.5	6.8
SCS	0.064	0.058	0.163	0.095	10.5	1.0

Table 7.9: CSG Tool Rendering Performance



(a) Blank
$b_1 \cup b_2 \cup b_3$

(b) Swept Volumes
$r_1 \cup \cdots \cup g_1 \cdots$

(c) Subtracted Result
$b_1 \cdots - r_1 \cdots - g_1 \cdots$

Figure 7.7: CSG Step Tool Model

7.3.7 Step Tool

The step tool model is a more elaborate and realistic version of the tool model in the previous section. Rather than a simple cylindrical blank, volumes are subtracted from a blank with multiple cylindrical and tapered sections. This model is also representative of an intended application of the SCS algorithm — tool grinding simulation.

The CSG tree illustrated in Figure 7.7 is formed by the sections of the blank: $b_1 \cup b_2 \cup b_3$, with red swept volumes: $r_1 \cup r_2 \cup \cdots$ and green swept volumes: $g_1 \cup g_2 \cup \cdots$ subtracted. In normalised form the tree is three CSG products:

- $b_1 - r_1 - r_2 - \cdots - g_1 - g_2 - \cdots$
- $b_2 - r_1 - r_2 - \cdots - g_1 - g_2 - \cdots$
- $b_3 - r_1 - r_2 - \cdots - g_1 - g_2 - \cdots$

Each product corresponds to a section of the blank with all of the swept volumes subtracted. Overlap graph processing reduces the size of each product by eliminating swept volume elements not overlapping the blank segment. Further detail of this example is given in Section 4.4.2.

Rendering Algorithm	time (s/frame)			average		relative
	GF3	GF4	FX	(s/frame)	(Hz)	time
Goldfeather	9.840	9.175	8.752	9.256	0.1	20.9
Layered Goldfeather	5.287	4.938	4.742	4.989	0.2	11.3
Imp. Layered Goldfeather	1.891	1.693	2.715	2.100	0.5	4.7
SCS	0.417	0.401	0.511	0.443	2.3	1.0

Table 7.10: CSG Step Tool Rendering Performance

From the results in Table 7.10, the SCS algorithm achieves the best average frame rate of 2.3 Hz, compared to 0.1–0.5 Hz for the Goldfeather implementations. This demonstrates the performance advantage of the SCS algorithm. It also illustrates that CSG rendering of these kinds of models is not quite possible at interactive frame rates. Frame rates below 10 Hz are generally not considered sufficient for interactive applications.

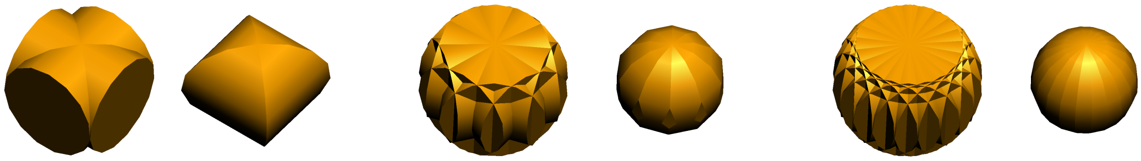
7.4 SCS Performance

The previous section demonstrates performance advantages of the SCS CSG rendering algorithm, in comparison to previously reported approaches. This section focuses on performance aspects of the SCS algorithm itself. Several features of the algorithm are beneficial to overall performance, depending on the situation.

Section 7.4.1 examines the linear-time intersection of convex objects, which is one of the unique characteristics of the SCS approach. Section 7.4.2 quantifies the performance advantage of using intersection in preference to subtraction for interactive CSG rendering of the Swiss cheese model introduced in Section 7.3.5.

Section 7.4.3 and Section 7.4.4 focus on overlap graph subtraction sequences. Overlap graph processing often results in shorter subtraction sequences and consequently better performance overall. However, this overall improvement depends on the number and arrangement of edges in the overlap graph, and the cost of the overlap graph processing step.

One of the simplifying assumptions made by the SCS algorithm is that concave geometry is handled only indirectly by the algorithm. Concave shapes need to be converted into convex geometry for the purpose of rendering with the SCS algorithm. Section 7.4.5 compares the real-world performance of rendering convex geometry with the SCS algorithm with methods that handle concave geometry natively.



(a) 2 Cylinders (b) 5 Cylinders (c) 10 Cylinders

Figure 7.8: CSG Intersected Cylinders

7.4.1 SCS Intersection

The SCS CSG rendering algorithm uses a linear-time algorithm for intersecting convex primitives in a product. The algorithm is described in detail in Section 4.2.1. This section experimentally confirms the linear relationship using a generalisation of the Intersected Cylinders model in Section 7.3.4.

The results were obtained on a 1Ghz Pentium III system with Nvidia GeForce3 OpenGL graphics, running RedHat Linux 7.1. Images were rendered in a 1024x768 window, each `gluCylinder` having 15 slices and 10 stacks.

The performance of SCS intersection averaged over 1000 frames appears as a table in Figure 7.9. In each run, a product of n intersected cylinders form a faceted spherical shape, as illustrated in Figure 7.8. The graph shows the linear relationship between the rendering time per frame, and the number of intersected cylinders.

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

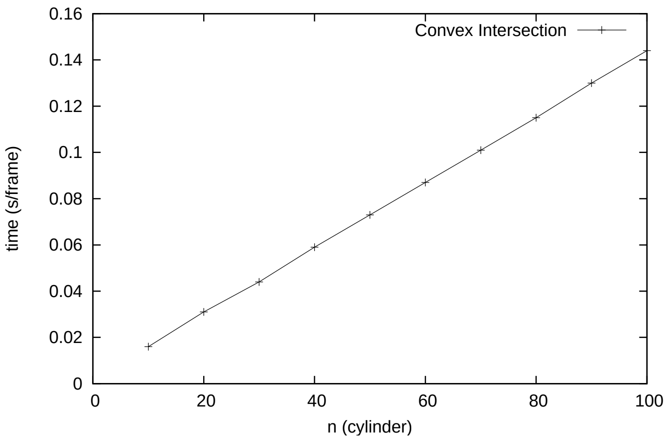
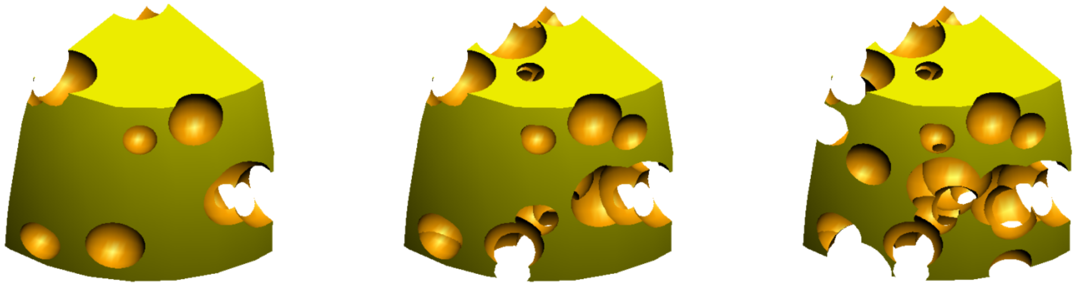


Figure 7.9: Cylinder Intersection Performance



(a) 25 subtracted spheres

(b) 50 subtracted spheres

(c) 75 subtracted spheres

Figure 7.10: Swiss Cheese Model

7.4.2 Intersection versus Subtraction

This experiment demonstrates the advantage of using intersection rather than subtraction in a CSG tree. Our motivation is that intersection of convex objects is linear-time, while subtraction is not.

A test model was developed for verifying the performance advantage of the SCS intersection. The Swiss Cheese CSG model in Figure 7.10 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: $ellipsoid - box_1 - box_2 - box_3 - box_4 - hole_1 - hole_2 - \cdots - hole_n$.

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

The average time per frame and frame rates for the two CSG trees are given in Table 7.11. Time per frame is plotted in Figure 7.11.

The non-linear time requirement for subtraction is evident in the increasing slope of both plots. The relative performance of intersection and subtraction can be contrasted to some extent by comparing convex intersection in Figure 7.9 and subtraction in Table 7.11. For $n = 100$, there is a considerable slowdown from 7 Hz for intersection to 1.2 Hz for subtraction.

The relative efficiency of intersection is also evident by contrasting 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

n	Subtraction		Intersection and Subtraction		
	Time (s/frame)	Frame Rate (Hz)	Time (s/frame)	Frame Rate (Hz)	Speedup (%)
10	0.069	14	0.050	20	29
20	0.111	9.0	0.078	13	27
30	0.164	6.1	0.120	8.3	25
40	0.227	4.4	0.169	5.9	26
50	0.294	3.4	0.217	4.6	24
60	0.357	2.8	0.286	3.5	20
70	0.455	2.2	0.370	2.7	20
80	0.588	1.7	0.500	2.0	15
90	0.714	1.4	0.588	1.7	17
100	0.833	1.2	0.714	1.4	15

Table 7.11: Swiss Cheese With and Without Intersection.

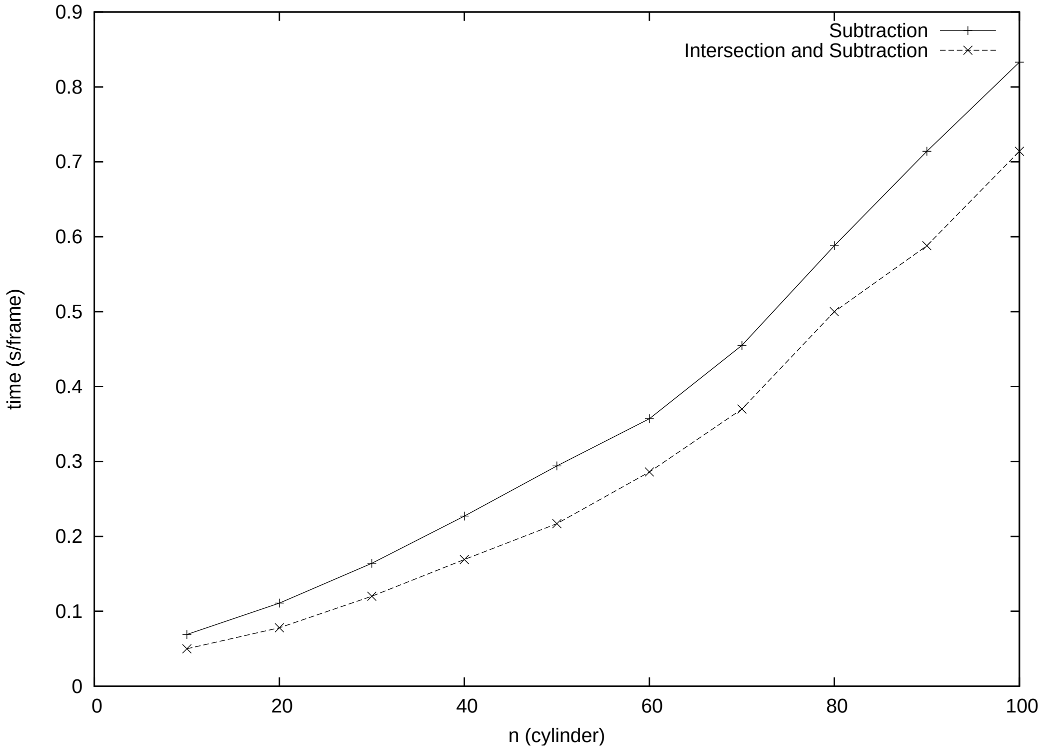
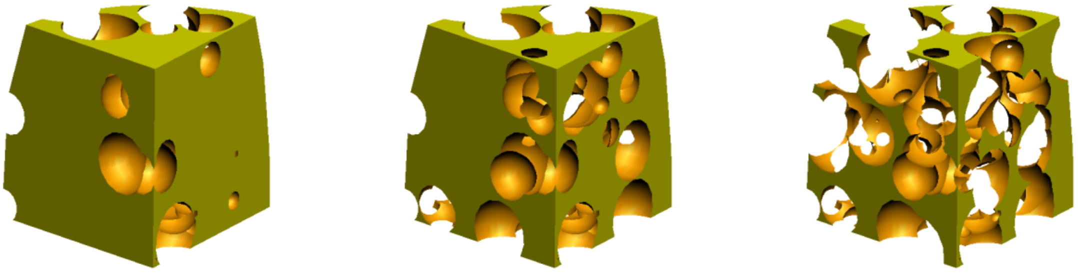


Figure 7.11: Swiss Cheese With and Without Intersection.

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 appropriate use of the convex intersection algorithm results in a performance improvement over a purely subtractive algorithm. They also suggest that intersection should be used (or even substituted) in preference to subtraction wherever possible.



(a) n=50 (b) n=100 (c) n=200

Figure 7.12: Procedural Swiss Cheese

7.4.3 Overlap Graph Subtraction Sequences

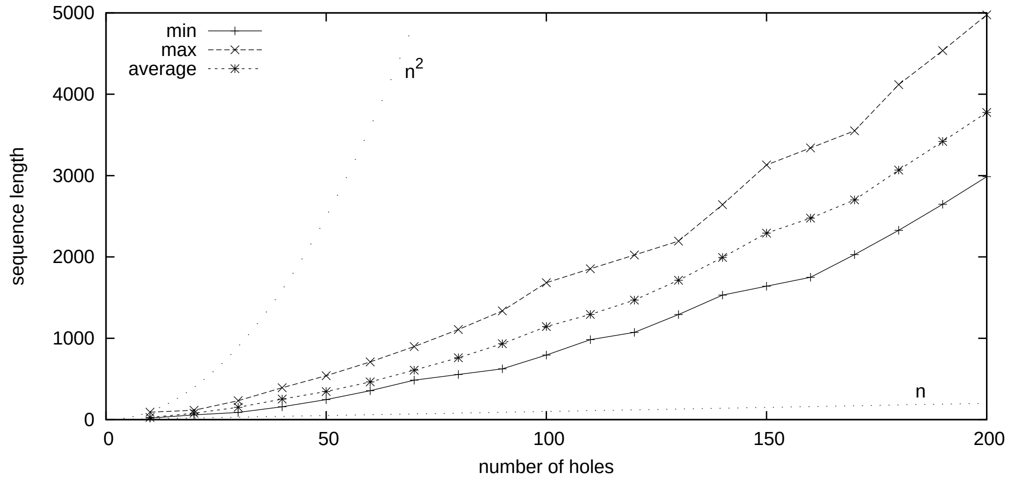
The Swiss cheese CSG model is generated procedurally. The user supplies a specification of the number and maximum size of holes, and a seed for the random number generator. In this experiment, the maximum size of holes and random number sequence are fixed, and the number of holes varied between 10 and 200. The model is illustrated in Figure 7.12 with 50, 100 and 200 holes.

Table 7.12 lists the minimum, maximum and average subtraction sequence lengths for Swiss cheese with up to two hundred holes. The sequence lengths were obtained by observing one thousand random viewing directions. kPES lengths are plotted in Figure 7.13(a), and overlap graph object-space sequence lengths are plotted in Figure 7.13(b). The improvement of the overlap graph object-space sequence over kPES for the average sequence length is plotted as an improvement factor in Figure 7.13(c). The improvement factor asymptotes to 1.0 as the number of holes increases.

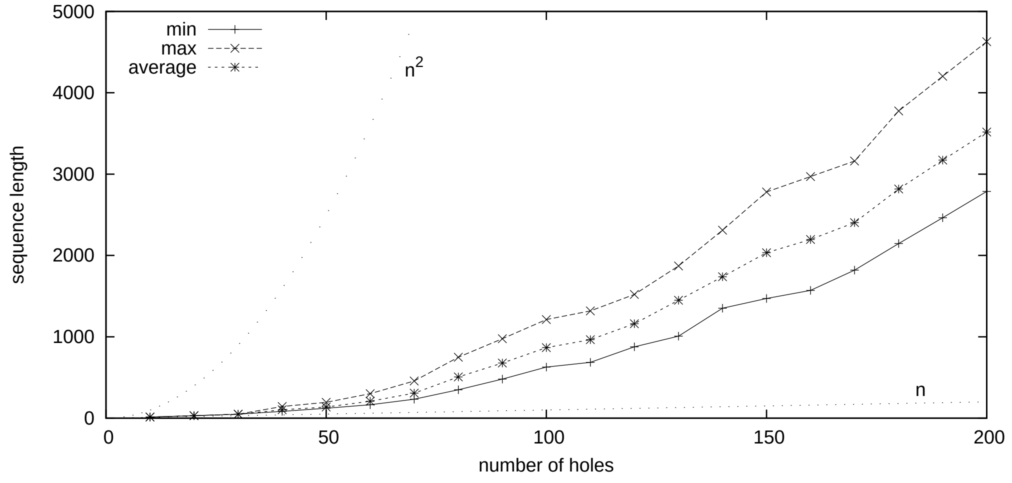
The data was collected in increments of $n = 10$ as plotted in Figure 7.13. Increments of $n = 20$ appear in Table 7.12 for the sake of brevity.

n	kPES			Overlap graph sequence		
	min	max	average	min	max	average
20	58	115	78	30	30	30
40	157	391	252	83	143	103
60	355	709	463	164	301	208
80	554	1107	760	349	749	506
100	793	1684	1143	628	1212	867
120	1072	2024	1468	876	1520	1160
140	1530	2642	1993	1351	2311	1738
160	1750	3340	2476	1570	2970	2197
180	2328	4118	3069	2146	3776	2817
200	2986	4976	3777	2788	4628	3517

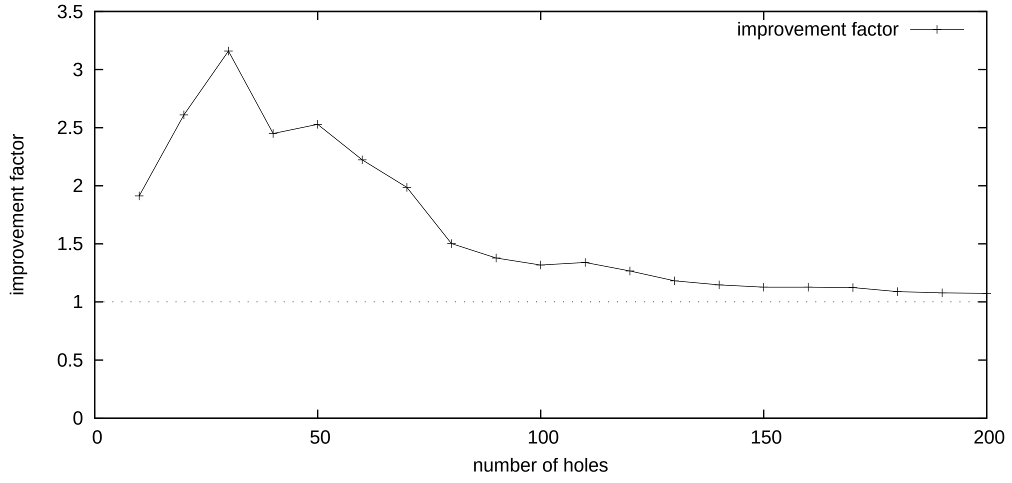
Table 7.12: Swiss Cheese Sequence Length



(a) *k*PES subtraction sequences



(b) Overlap graph subtraction sequences



(c) Overlap Graph improvement factor

Figure 7.13: Swiss Cheese Subtraction Sequences

Overlap graphs for $n = 25$, 50 and 100 are illustrated in Figures 7.14, 7.15 and 7.16. Dotted nodes indicate those with no spatial overlap (Section 6.2.6) and so appear just once in the final subtraction sequence. Nodes processed in leaf trimming passes (Section 6.2.3) appear unshaded and appear twice in the final subtraction sequence. Shaded nodes forming ring subgraphs (Section 6.2.4) appear up to four times in the final subtraction sequence. The remaining shaded nodes are those forming cyclic subgraphs, each of which is encoded into an $O(kn)$ or $O(n^2)$ subtraction sequence.

These results indicate that object-space overlap graph subtraction sequences are consistently shorter, with minimum, maximum and average case characteristics all being similar. In Figure 7.13(c) for small n, sequence length improvement factors of between two and three are observed until $n = 40$ when leaf trimming opportunities diminish. As n approaches 80, the relative advantage drops dramatically and approaches an asymptote at unity as n increases. This reflects the fact that as the number of holes increases the overlap graph becomes more dense and cyclic and the opportunity for leaf node pruning diminishes. Overlap graph subtraction sequences will never be longer, since in the worst case the image-space encoding algorithm is used.

As the number of nodes is increased, the proportion of cyclic nodes also increases. For $n = 25$ in Figure 7.14 there is only one cyclic subgraph, compared to five cyclic subgraphs for $n = 50$ in Figure 7.15. A30, A39 and A37 are responsible for the formation of the two major cyclic subgraphs illustrated in Figure 7.15 for $n = 50$. The resulting decline in improvement factor for $n = 40$ compared to $n = 30$ is evident in Figure 7.13(c).

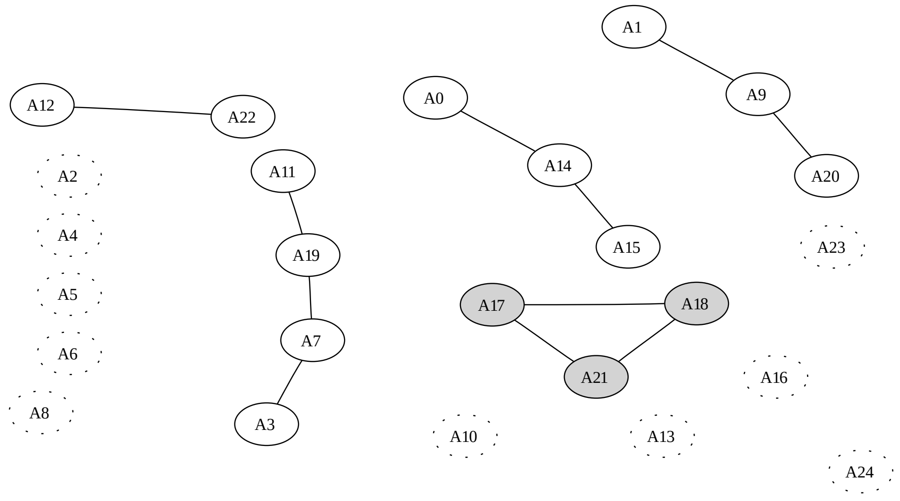


Figure 7.14: Swiss Cheese Overlap Graph for $n = 25$

As the number of nodes is further increased, these cyclic subgraphs merge into one large interconnected component illustrated in Figure 7.16. Relatively few nodes can be processed in leaf-trimming passes, leaving the majority to be encoded as an $O(kn)$ or $O(n^2)$ subtraction sequence. Subtraction sequences therefore trend from a linear to quadratic length subtraction sequence as nodes are added.

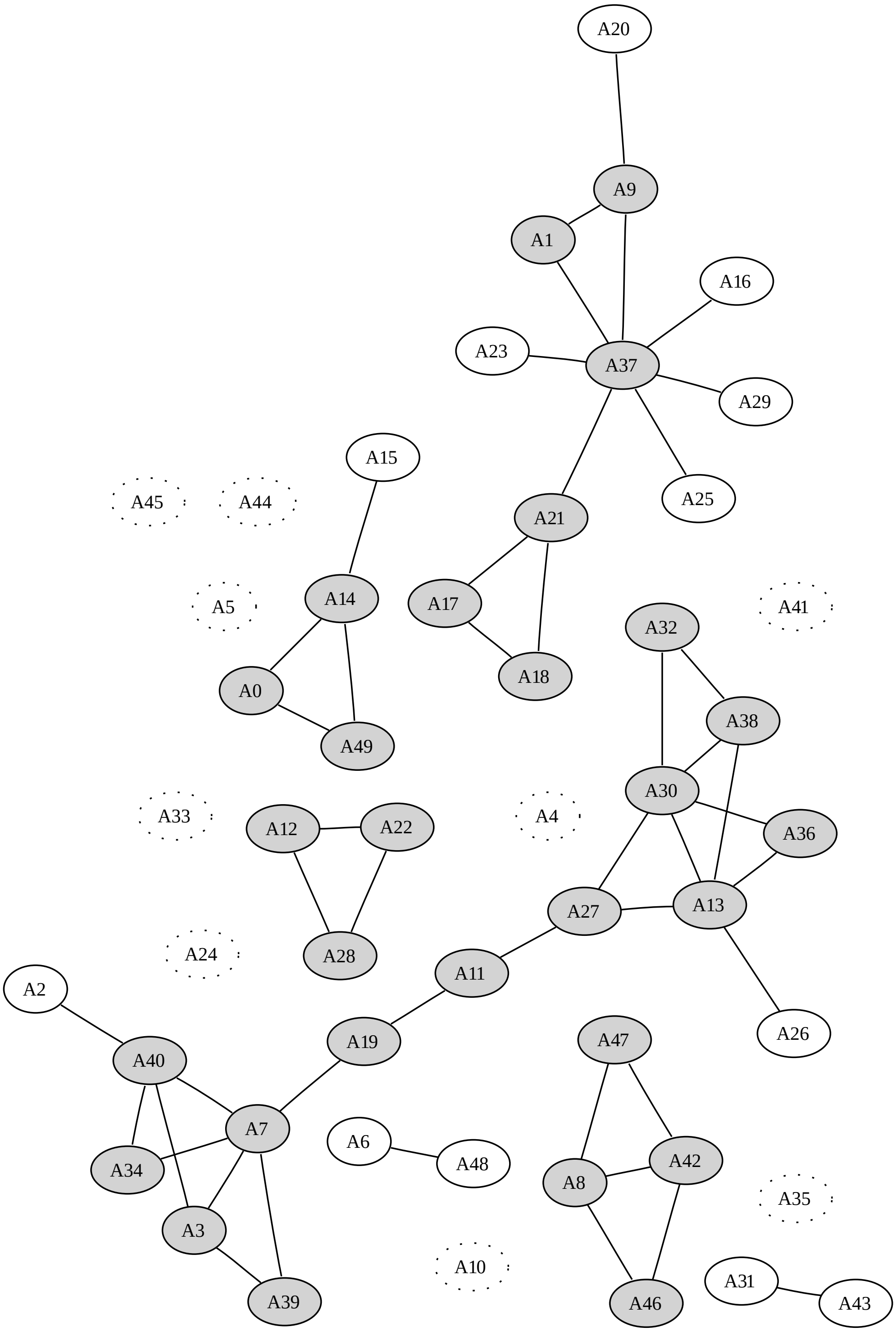


Figure 7.15: Swiss Cheese Overlap Graph for $n = 50$

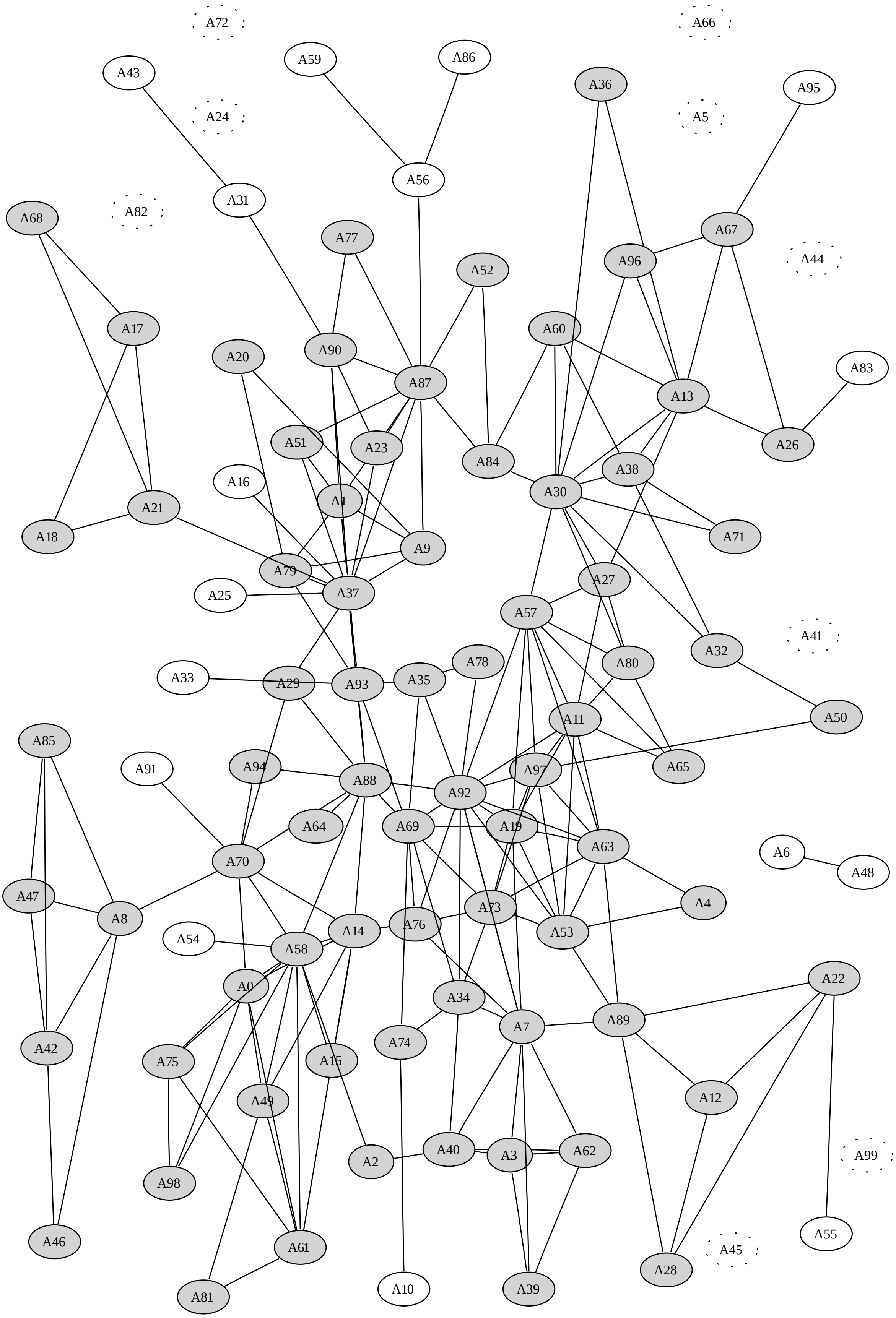
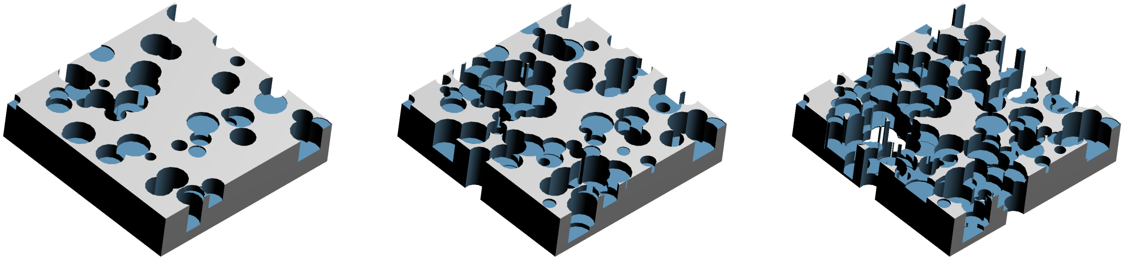


Figure 7.16: Swiss Cheese Overlap Graph for $n = 100$



(a) n=50 (b) n=100 (c) n=200

Figure 7.17: Simulated Three Axis Drilling

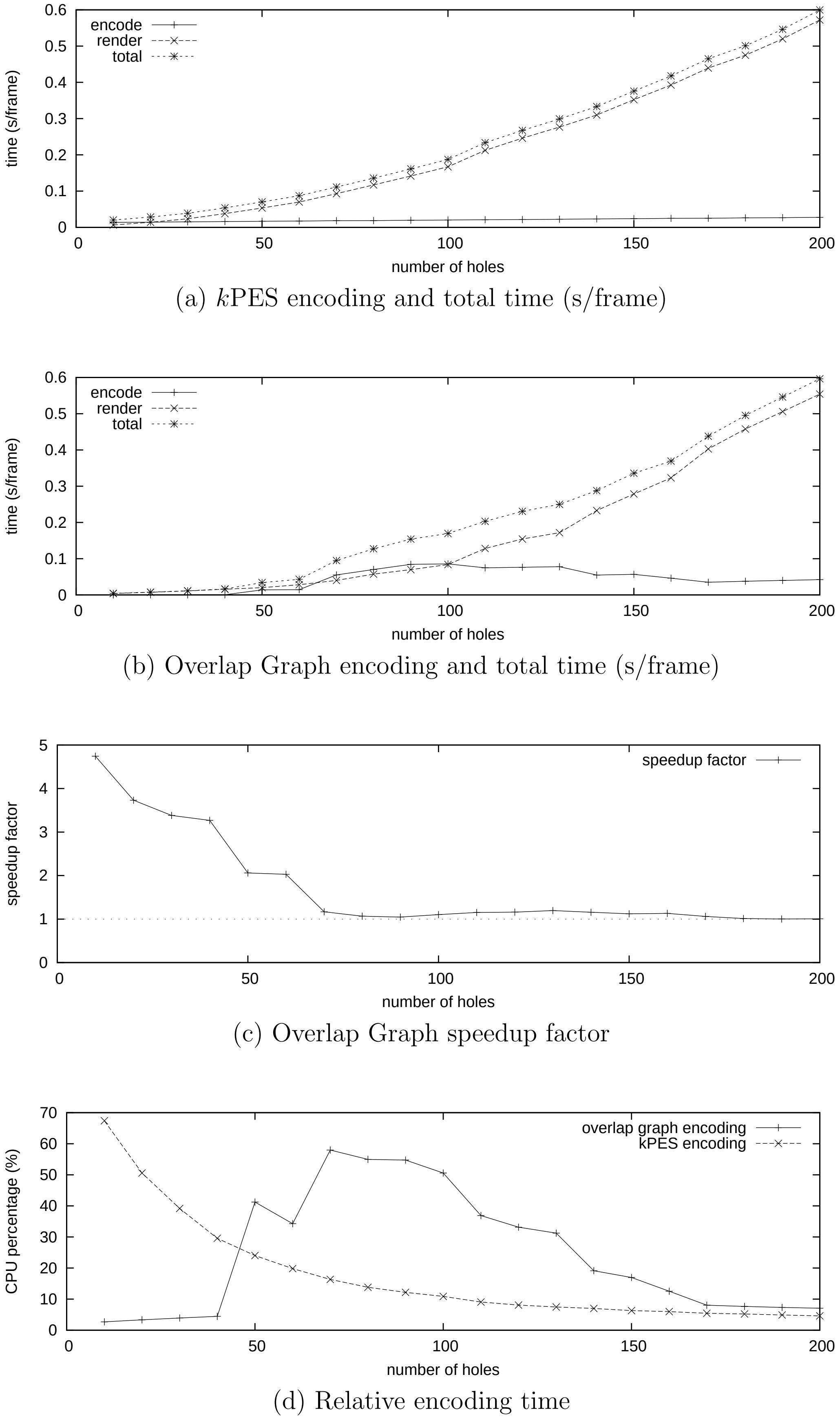
7.4.4 Overlap Graph Rendering Time

This experiment considers a simulated three axis drilling scenario. User specified parameters for the number and maximum size of holes are used to randomly generate drilled holes of varied radius and position. The model is illustrated in Figure 7.17 with 50, 100 and 200 holes.

This experiment focuses on time rather than sequence length and examines the processing overhead of object-space overlap graph sequence encoding. Table 7.13 lists the encoding time and total time for three axis drilling models with up to two hundred holes. One thousand random viewing directions were sampled for *k*PES and overlap graph based sequence encoding methods. *k*PES rendering times are plotted in Figure 7.18(a) and overlap graph rendering times are plotted in Figure 7.18(b). The relative performance of overlap graph CSG rendering is plotted as a speedup factor in Figure 7.18(c). The proportion of time spent encoding subtraction sequences is plotted as a fraction in Figure 7.18(d).

	*k*PES (s/frame)			Overlap Graph (s/frame)		
n	encode	render	total	encode	render	total
20	0.0144	0.0141	0.0285	0.0003	0.0074	0.0076
40	0.0159	0.0379	0.0538	0.0007	0.0157	0.0164
60	0.0173	0.0700	0.0873	0.0148	0.0283	0.0430
80	0.0187	0.1171	0.1358	0.0701	0.0574	0.1274
100	0.0203	0.1669	0.1873	0.0858	0.0839	0.1697
120	0.0216	0.2458	0.2674	0.0765	0.1542	0.2307
140	0.0233	0.3101	0.3333	0.0551	0.2329	0.2880
160	0.0250	0.3926	0.4176	0.0464	0.3230	0.3693
180	0.0261	0.4748	0.5009	0.0378	0.4576	0.4954
200	0.0276	0.5719	0.5995	0.0422	0.5541	0.5963

Table 7.13: Three Axis Rendering Time (s/frame)



(a) kPES encoding and total time (s/frame)

(b) Overlap Graph encoding and total time (s/frame)

(c) Overlap Graph speedup factor

(d) Relative encoding time

Figure 7.18: Three Axis Rendering Time

For n up to 40, all nodes are processed in leaf trimming making depth complexity sampling unnecessary. Overlap graph sequence encoding is particularly advantageous in these cases, resulting in overall speedups of between three and four. As drill holes start forming cyclic clusters for $60 \leq n \leq 90$, execution time for overlap graph encoding increases in both absolute and relative terms. For $n > 100$ overlap graph encoding time decreases as the model becomes increasingly dense and cyclic. As opportunities for leaf trimming and ring finding diminish, the overall performance of both approaches converge.

For overlap graph sequence encoding, the fraction of time spent encoding the subtraction sequence peaks at nearly 60%, but still results in an overall speedup. In these experiments, the per-frame overlap graph encoding time is always more than offset by the performance benefit of shorter subtraction sequences. Stencil buffer copying related to depth complexity sampling has been observed to be a bottleneck at resolutions higher than 800x600. The size at which stencil buffer copying becomes the bottleneck depends on the relative rasterisation and buffer copying performance of a particular platform.

Overlap graphs for $n = 25$, 50 and 100 are illustrated in Figures 7.19, 7.20 and 7.21. As in the previous section dotted nodes indicate those with no spatial overlap (Section 6.2.6) and appear once in the final subtraction sequence. Nodes processed in leaf trimming passes (Section 6.2.3) appear unshaded and appear twice in the final subtraction sequence. Shaded nodes forming ring subgraphs (Section 6.2.4) appear up to four times in the final subtraction sequence. The remaining shaded nodes are those forming cyclic subgraphs, each of which is encoded into an $O(kn)$ or $O(n^2)$ subtraction sequence.

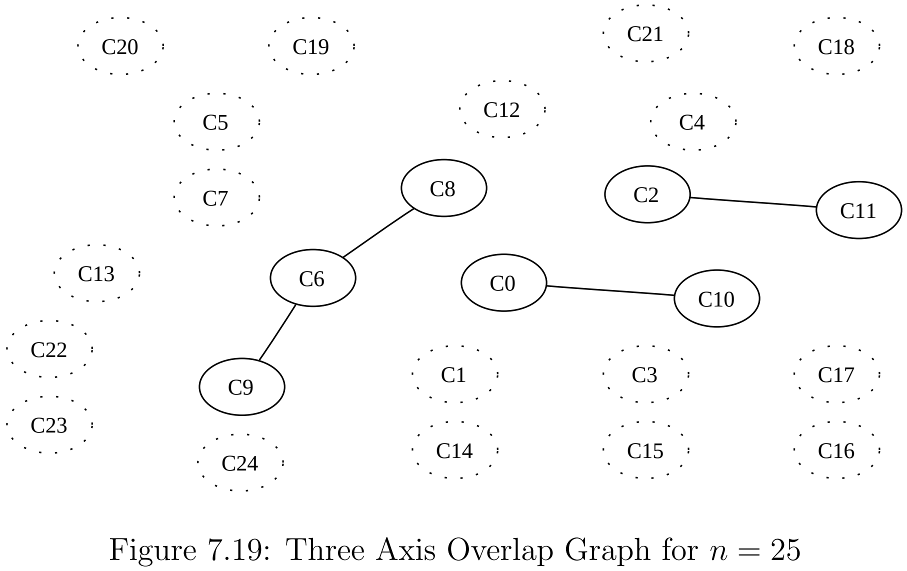


Figure 7.19: Three Axis Overlap Graph for $n = 25$

Compared to the overlap graphs in the previous section, the three axis scenario results in a similar pattern of increasing cyclicity. Although the drilled holes are constrained to two dimensions instead of three, individual holes are somewhat less likely to overlap spatially because they are relatively small.

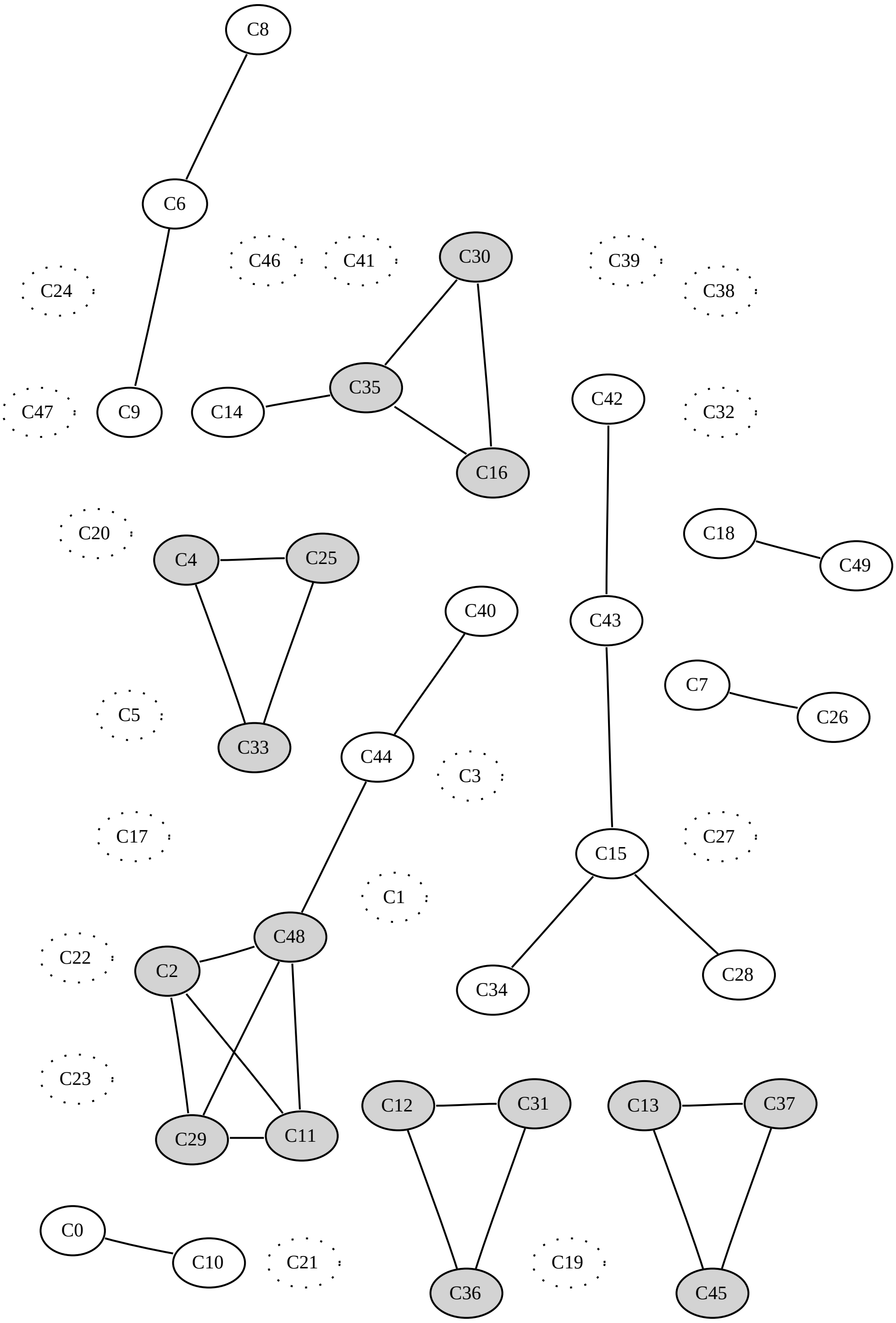


Figure 7.20: Three Axis Overlap Graph for $n = 50$

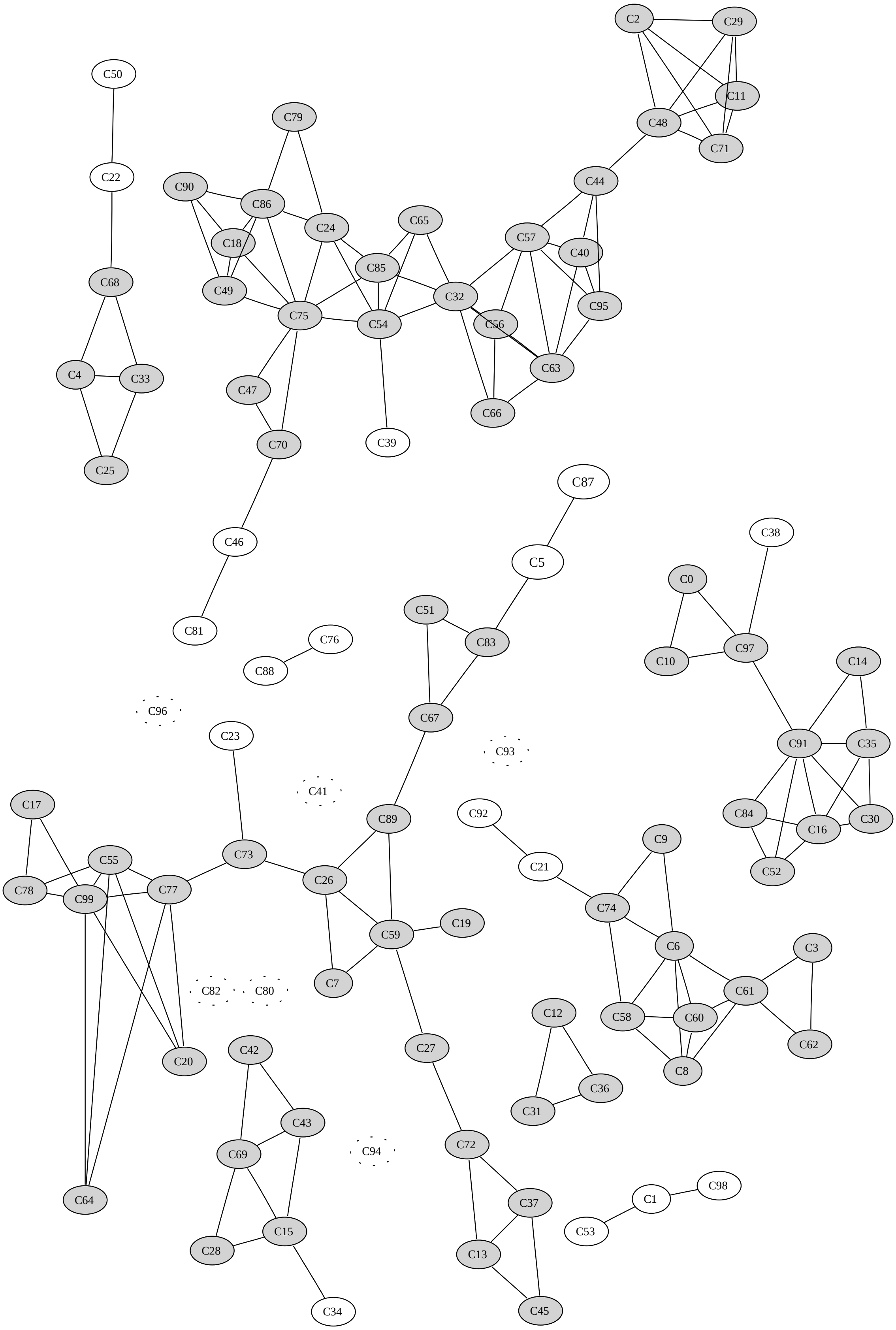
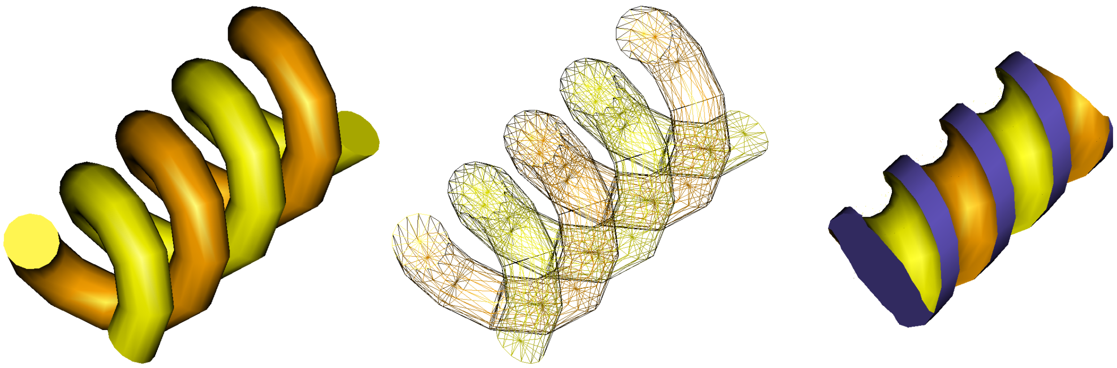


Figure 7.21: Three Axis Overlap Graph for $n = 100$



(a) Concave helix shape (b) Convex decomposition (c) Subtracted result

Figure 7.22: Convex and Concave Helical Volume Representation

7.4.5 Convex versus Concave Shapes

The SCS algorithm is specifically designed to process CSG trees of convex shapes, whereas the Goldfeather algorithm can also handle concave shapes. SCS handles concave shapes by requiring a decomposition into convex shapes.

This experiment examines the performance of CSG rendering algorithms using either convex or concave representations. Tesselating concave volumes into convex elements results in a larger number of volumes and a larger surface area for processing. However, convex elements allow algorithmic simplifications — each object can be processed at a lower cost with convex assumptions. This same approach of convex decomposition is used in OpenGL for handling concave polygons.

The model used in this experiment is illustrated in Figure 7.22. Two helical volumes are subtracted from a cylinder to form a cutting or milling tool. Each helix is either represented as (a) a concave volume, or (b) decomposed into convex segments, to form (c) the subtracted result.

The platform used for timing the algorithms is an AMD Athlon XP 1800+ with 256 kB cache, 1GB RAM using Nvidia FX 5900 Ultra graphics card running the Fedora Core 3 Linux operating system and Nvidia version 71.74 OpenGL drivers. The SCS and Goldfeather CSG rendering time was averaged over at least 1000 frames, in an 800x600 window.

	x	b_1	c_1	d_1	e_1	f_1	g_1	h_1	i_1	j_1	k_1	l_1	m_1	n_1	o_1	p_1	q_1	r_1	s_1	t_1	u_1
x	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
b_1	●	●	●																		
c_1	●	●	●	●																	
d_1	●		●	●	●																
e_1	●			●	●	●															
f_1	●				●	●	●														
g_1	●					●	●	●													
h_1	●						●	●	●												
i_1	●							●	●	●											
j_1	●								●	●	●										
k_1	●									●	●	●									
l_1	●										●	●	●								
m_1	●											●	●	●							
n_1	●												●	●	●						
o_1	●													●	●	●					
p_1	●														●	●	●				
q_1	●															●	●	●			
r_1	●																●	●	●		
s_1	●																	●	●	●	
t_1	●																		●	●	●
u_1	●																			●	●

Table 7.14: Helix Overlap Matrix

The overlap matrix in Table 7.14 represents the spatial overlap between the cylinder x and the convex elements $b_1, c_1, d_1, \cdots, u_1$ of one of the helices. Each helix element is touching one or two others, as well as the cylinder. These subtracted elements form an acyclic graph making it well suited for graph based sequence encoding and resulting in an $O(n)$ subtraction sequence. The second helix in the model also has the same overlap characteristics, being rotated with respect to the axis of the cylinder. It should be noted that if the two helices were overlapping each other, the overlap graph of the subtracted elements would no longer be acyclic.

Timing results for the convex and concave versions of the model in Table 7.15 show the performance advantage of the SCS approach for this model. The SCS algorithm can render at 152 Hz, compared to the 20 Hz of the Goldfeather algorithm. The convex representation is particularly poorly suited for the Goldfeather approach, resulting in less than 6 Hz.

Algorithm		average		relative
		(s/frame)	(Hz)	time
Convex	Goldfeather	2.110	0.5	301.4
	Layered Goldfeather	0.281	3.6	40.1
	Imp. Layered Goldfeather	0.174	5.8	24.9
	SCS	0.007	152.1	1.0
Concave	Goldfeather	0.050	19.9	7.1
	Layered Goldfeather	0.166	6.0	23.7
	Imp. Layered Goldfeather	0.076	13.0	10.0
	SCS	n/a	n/a	

Table 7.15: Convex and Concave CSG Rendering Performance

7.5 Discussion

To conclude this chapter, performance aspects of the SCS CSG rendering algorithm are summarised, and some assumptions and limitations discussed.

The SCS CSG algorithm achieves interactive frame rates on commodity PC hardware platforms for some of the models, but not the more complicated ones. Depth buffer copying has been identified as a significant bottleneck for CSG rendering, motivating the SCS algorithm which makes heavier utilisation of rasterisation, rather than buffer copying. The algorithm has been shown to be advantageous for a variety of CSG models, compared to the Goldfeather-style algorithms.

Several performance aspects of the SCS CSG algorithm are investigated experimentally and the results discussed. Intersection of convex objects is performed in linear time. Subtraction of convex objects is linear time in the best case, $O(n^2)$ time in the worst case. Overlap graph processing can utilise spatial overlap information to reduce the length of subtraction sequences, especially in cases of low to moderate overlap. Handling of concave shapes as convex decompositions can result in an overall speedup, compared to algorithms that process concave shapes directly.

The SCS CSG rendering algorithm was formulated, developed and tested for fixed-pipeline OpenGL graphics hardware. A variety of Nvidia graphics hardware was used in this work, other vendors such as ATI were not tried. With the rapid pace of PC and graphics hardware we would expect CPU speed, GPU performance and bus bandwidth to affect the relative performance of CSG rendering algorithms.

The CSG models used in this work tend to be test-cases rather than real-world data sets, although some of the models are intended to be similar to real-world cases. Developing a more comprehensive set of models to represent real world examples is a suggestion for further work.

Chapter 8

Conclusion

If a man will begin with certainties, he shall end in doubts; but if he will be content to begin with doubts he shall end in certainties.
— Sir Francis Bacon

8.1 Conclusion

This work has introduced a new approach to CSG rendering based on sequenced subtraction of convex objects on graphics hardware. We have called this *Sequenced Convex Subtraction*[1] (SCS), drawing on the Trickle algorithm [28] concept of front to back subtraction of convex layers, but utilising convex objects instead of frame-buffer layers and utilising a subtraction sequence instead of sorting by depth.

The SCS algorithm is motivated by a specific application domain and performance characteristics of fixed-pipeline graphics hardware. The algorithm has been found to be applicable to a broad set of CSG models[2], although is better suited to some situations than to others. The SCS approach is particularly well suited to models involving convex objects with little to moderate overlap, and moderate overall depth complexity[3]. The algorithm is suitable for implementation in mainstream graphics hardware, and is designed to minimise the number of buffers and buffer copying required by prior approaches.

This work has introduced the concept of a subtraction sequence, the specific sequence in which convex objects are subtracted from the depth buffer. A subtraction sequence can be derived from information about the subtracted objects including the number of objects, the view-dependent depth complexity of the objects and the pairwise overlap of the objects. In the general case an $O(n^2)$ sequence is used[4]. If depth complexity k is known, an $O(kn)$ sequence can be used[5]. Utilisation of pairwise overlap information can result in $O(1)$ or $O(n)$ sequences in particular circumstances[6].

Subtraction sequences have also been examined from the abstract mathematical viewpoint of Permutation Embedding Sequences[7] (PESs). Previous work [14, 35, 83] has established algorithms for generating $n^2 - 2n + 4$ length subtraction sequences, but has not established a lower bound for PES length.

This work introduces the concept of Normalised No Repeat (NNR) sequences, a superset of the shortest length subtraction sequences that serves to reduce the overall space of possible subtraction sequences. Utilising NNR we were able to determine a complete list[8] of shortest length subtraction sequences for $1 \leq n \leq 5$, which has not been reported

[1] Chapter 4 and 6.
[2] Chapter 7.
[3] Section 7.4.3.
[4] Chapter 5.
[5] Section 4.3.2.
[6] Chapter 6.
[7] Chapter 5.
[8] Table 5.7.

previously. We did not discover any PESs of shorter length than $n^2 - 2n + 4$. The NNR approach also establishes an $\Omega(n^2)$ lower bound for PES length[9], which has not been reported previously.

This work also examined the utilisation of pairwise subtracted object overlap for subtraction sequences[10]. Non-cyclic parts of the overlap graph can be utilised to obtain $O(n)$ subtraction sequences, and certain cyclic sub-graphs can also result in $O(n)$ sequences. Experimentally, we have shown that the performance improvement resulting from overlap graph subtraction sequences is worth the CPU computational cost, even in fairly degenerate worst cases[11].

This work has also experimentally confirmed the correctness and performance of the SCS approach to CSG rendering by means of an implementation[12] in C++ and OpenGL, and application to a variety of CSG models[13].

We are optimistic about hardware and algorithmic progress towards CSG rendering using graphics hardware and the potential for useful interactive applications of CSG modelling.

[9] Section 5.4.7.

[10] Chapter 6.

[11] Section 7.4.4.

[12] Appendix B and available online.

[13] Chapter 7.

8.2 Further Work

There are a variety of future directions for hardware based CSG rendering. We will mention here some of the unresolved questions and potential improvements to the SCS approach in particular.

Graphics hardware presents a “moving target” in terms of features and performance characteristics. At this point in time the technology horizon for interactive computer graphics includes multi-core CPU, universal GPU shaders and integration of the CPU and GPU. SCS is formulated in terms of the OpenGL 1 fixed-pipeline architecture, with an emphasis on minimising depth buffer copying. The relative competitiveness of hardware-based algorithms is likely to shift over time. New graphics hardware features may be better suited to a particular CSG rendering approach.

We think there are further improvements that can be made to overlap graph subtraction sequences. Taking into account the viewing direction and the depth relationship of overlapping objects, the graph could be treated as a directed graph, greatly improving the potential for non-cyclic leaf trimming, resulting in shorter subtraction sequences. Nodes could be grouped into clusters and treated as non-cyclic leaves, resulting in $O(\frac{2n^2}{g})$ sequences, being especially beneficial when the groups are of comparable size and the number of groups g is greater than 2.

PESs of shortest length remains an unsolved problem. Some ideas for further investigation of these were mentioned previously in Section 5.5.1.

Finally, there is a broader question of the “ideal” hardware-based CSG rendering approach. To what extent can electronics advances, graphics hardware design, engineering and manufacturing solve the problem? Alternatively, is the CSG rendering challenge an algorithmic one of utilising graphics hardware and CPU in an optimal arrangement?

8.3 Citations and Subsequent Work

In the time during and after the development of the SCS algorithm there has been other related work reported:

- The *Improved Layered Goldfeather Algorithm* [29] made better utilisation of the stencil buffer for Goldfeather CSG rendering. We have described this briefly in Section 3.5.3, and include experimental performance results for our implementation in Section 7.3.
- A survey of depth buffer techniques [93] includes discussion of CSG rendering, including SCS.
- CSG rendering using a two-sided depth test [42] improved layer-peeling performance by utilising the hardware shadow buffer depth test.
- Several SCS performance improvements were reported by utilising off-screen rendering as an alternative to depth or stencil copying and using the OpenGL occlusion query for depth complexity [52].
- The OpenCSG real-time 3D graphics library [53] implements CSG rendering using algorithms including SCS.
- The Blister CSG rendering algorithm [43] avoids conversion of the CSG to sum-of-products form and utilises hardware-based layer peeling.
- Subsequent to the Blister approach, the *Constructive Solid Trimming* (CST) algorithm [46] and Optimised Blist Form [76].
- Transparent and edge-enhanced interactive CSG rendering [67].
- A surfel approach to CSG rendering [13].
- Hardware-based slicing algorithm for CSG [58, 44].
- Romeiro et al. [33] report spatial subdivision, CSG tree simplification shader-based raytracing of CSG models.

Appendix A

Software Documentation

I would sooner read a timetable or a catalog than nothing at all.

— W. Somerset Maugham

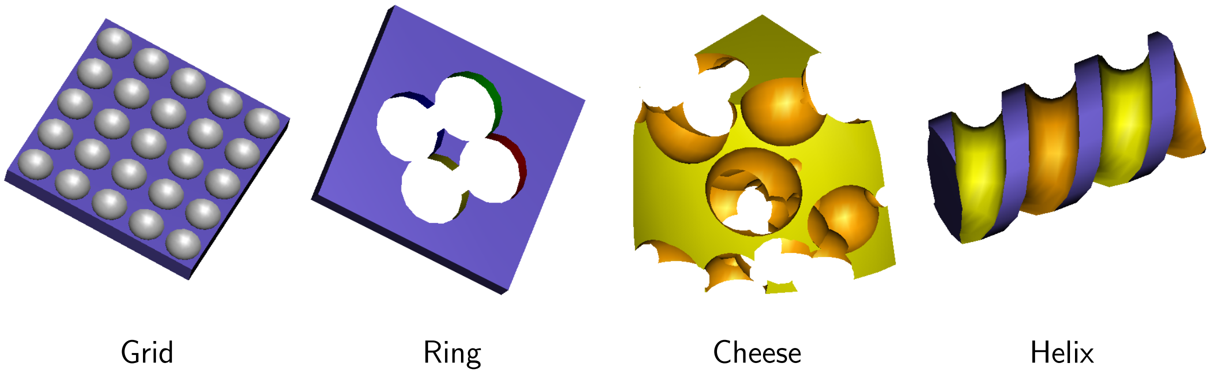


Figure A.1: CSG Models Included with CsgDemo

A.1 CsgDemo

A.1.1 Introduction

CsgDemo is an interactive real-time graphical application for demonstrating and experimenting with real-time OpenGL CSG rendering algorithms. The Goldfeather [37, 97], Layered Goldfeather [87], Improved Layered Goldfeather [29] and Sequenced Convex Subtraction [88, 89, 30] (SCS) CSG rendering algorithms are implemented. A number of CSG trees are included with the program, as well as the facility to import CSG models via a simple file format. Some included CSG models are illustrated in Figure A.1.

A.1.2 System Requirements

CsgDemo is available for Windows with OpenGL drivers installed and Linux with Mesa or OpenGL drivers. Hardware graphics acceleration is recommended. A variety of hardware platforms were used for development including 3Dlabs Permedia 2 and Nvidia TNT2, Geforce 2, GeForce 3, GeForce 4 and GeForce FX.

Left Mouse Button	Menu
Right Mouse Button	Rotation
Right Mouse Button + SHIFT	Panning
Right Mouse Button + CTRL	Zooming
1,2,$\cdots$	Select CSG Tree
1,2,$\cdots$ *+ SHIFT*	Select information mode
i,o	Zoom in and out
a,z,x,c	Rotation
g,b,n,m	Panning
f,v	Forward and back in z
[,],{,}	Scale and translate frustum in z
+,-	Adjust number of subtracted objects
F9	Save RGB colour buffer
F10	Save stencil buffer
F11	Save depth buffer
TAB	Toggle full-screen mode
q,ESC	Quit application

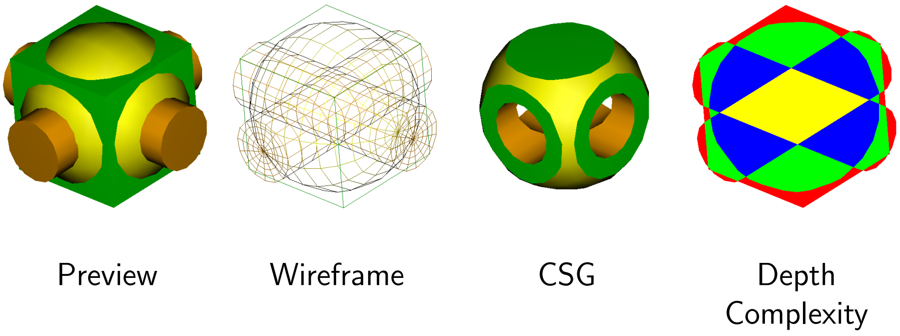
Table A.1: CsgDemo Mouse and Keyboard Bindings

A.1.3 Mouse and Keyboard

Interaction with CsgDemo is provided via mouse, keyboard and pop-up menus. The mouse and keyboard bindings are listed in Table A.1.

A.1.4 Menu

The CsgDemo pop-up menu is activated by clicking the left mouse button with the application window. The menu items are detailed in Table A.2. Several of the display modes are illustrated in Figure A.2.



Preview

Wireframe

CSG

Depth Complexity

Figure A.2: CsgDemo Display Modes

Model	Select CSG model for display.
Tree	Select and display CSG tree from current model.
Primitive	Select and display primitive from current model.
Algorithm	Select the rendering algorithm.
Preview	Display all surfaces without CSG rendering.
Wireframe	Display in wire-frame without CSG rendering.
Layer	Extract i^{th} layer. (Use l and L to adjust i)
Depth Complexity	Depth-complexity visualisation mode.
Goldfeather	Goldfeather [37, 97] CSG rendering algorithm.
Layered Goldfeather	Layered Goldfeather [87] CSG rendering algorithm.
Imp. Layered Goldfeather	Improved Layered Goldfeather [29] algorithm.
SCS	Sequenced Convex Subtraction [88, 89] algorithm.
Options	
Display	Options for configuring the 3D display.
Fast Manipulation	Faster display mode during mouse manipulation.
Bounding Boxes	Display bounding boxes of all objects.
Z Histogram	Display z-buffer histogram.
Benchmark Mode	Redraw continuously for benchmarking purposes.
Info	Information display options.
Introduction	Display title, version and credit information.
OpenGL Driver	Display OpenGL driver and version information.
CSG Information	Display CSG rendering statistics.
Overlap	Display current overlap matrix.
Frame Rate	Display frame-rate information.
Hide	No displayed information.
CSG Rendering	CSG Rendering options.
Copy/Draw	Use z-buffer copying or read/draw for multiple z-buffer.
Sample K	Use depth complexity sampling. (recommended)
Object-Space	Use object-space SCS subtraction sequence encoding.
BSP Overlap	Use Binary Space Partitioning for overlap graph.
Cull Faces	Use face culling.
Calculate Adjacency	Recalculate adjacency.
Save	Files are saved increasing numerically from `00000000.ext`
RGB	Save OpenGL colour buffer as PNG file.
Stencil Buffer	Save OpenGL stencil buffer as PPM file.
Z-Buffer	Save OpenGL depth buffer as PPM file.
Overlap (LaTeX)	Save overlap graph as LaTeXtable.
Overlap (dot)	Save overlap graph as dot [7] graph.

Table A.2: CsgDemo Menu Items

A.1.5 Command Line Options

CsgDemo supports a variety of command line options:

- `filename`

  Specify CSG model from an input file, or from standard input: `-`

- `-size width height`

  Specify the width and height of the window.

- `-ini filename`

  Specify a configuration file

- `-set "name:value"`

  Specify an internal setting.

- `-quiet`

  Suppress start-up messages.

- `+keys`

  Specify list of keyboard events for processing.

For example:

- `csgdemo -size 800 600`

  Run in an 800x600 window

- `csgdemo -fullscreen`

  Run in fullscreen mode

- `csgdemo +^`

  Show frame rate information

- `csgdemo -set csgdemo.shape.mode:DepthComplexity`

  Display depth complexity

- `csgdemo -set csgdemo.shape.mode:SCS`

  Display using the SCS CSG rendering algorithm

- `csgdemo -set csgdemo.display.benchmark:true -set csgdemo.samples:100`

  Draw 100 frames in benchmark mode

A.1.6 File Format

In addition to built-in CSG models, CsgDemo has the facility to read CSG models from a file or standard input. The CsgDemo file format is text based. Each line describes either a leaf shape, or a complete CSG tree. An example CSG model is illustrated in Figure A.3. The full grammar is as follows:

$\langle\text{file}\rangle \rightarrow \langle\text{entries}\rangle$

$\langle\text{entries}\rangle \rightarrow \langle\text{entry}\rangle \;\; \langle\text{entries}\rangle \;\; |$

$\langle\text{entry}\rangle \rightarrow \langle\text{leaf}\rangle \; | \; \langle\text{tree}\rangle \; | \; \langle\text{trans}\rangle \; |$

$\langle\text{leaf}\rangle \rightarrow \langle\text{name}\rangle \;\; = \;\; \langle\text{shape}\rangle \;\; \langle\text{color}\rangle \;\; \langle\text{transforms}\rangle$

$\langle\text{shape}\rangle \rightarrow \textbf{sphere} \; | \; \textbf{box} \; | \; \textbf{cylinder} \; | \; \textbf{helix} \; | \; \textbf{helixseg} \; | \; \textbf{mesh}$

$\langle\text{color}\rangle \rightarrow \textbf{black} \; | \; \textbf{white} \; | \; \textbf{red} \; | \; \textbf{green} \; | \; \textbf{blue} \; | \; \textbf{orange} \; | \; \textbf{yellow}$

$\langle\text{transforms}\rangle \rightarrow \langle\text{transform}\rangle \;\; \langle\text{transforms}\rangle \;\; |$

$\langle\text{transform}\rangle \rightarrow \langle\text{scale}\rangle \; | \; \langle\text{translate}\rangle \; | \; \langle\text{rotate}\rangle \; |$

$\langle\text{scale}\rangle \rightarrow \textbf{scale} \;\; \langle\text{vector}\rangle$

$\langle\text{translate}\rangle \rightarrow \textbf{translate} \;\; \langle\text{vector}\rangle$

$\langle\text{rotate}\rangle \rightarrow \textbf{rotate} \;\; \langle\text{vector}\rangle \;\; \langle\text{angle}\rangle$

$\langle\text{vector}\rangle \rightarrow \langle\text{number}\rangle \;\; \langle\text{number}\rangle \;\; \langle\text{number}\rangle$

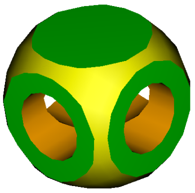
$\langle\text{angle}\rangle \rightarrow \langle\text{number}\rangle$

$\langle\text{tree}\rangle \rightarrow \textbf{Tree} = \;\; \langle\text{expression}\rangle$

$\langle\text{expression}\rangle \rightarrow \langle\text{name}\rangle \; | \; \textbf{(} \;\; \langle\text{expression}\rangle \;\; \langle\text{operator}\rangle \;\; \langle\text{expression}\rangle \;\; \textbf{)}$

$\langle\text{operator}\rangle \rightarrow \textbf{+} \; | \; \textbf{-} \; | \; \textbf{.}$

$\langle\text{trans}\rangle \rightarrow \textbf{Transform} = \;\; \langle\text{transforms}\rangle$



```
X1 = sphere yellow scale 2.0 2.0 2.0
X2 = box green scale 1.5 1.5 1.5
A = cylinder orange scale 0.8 0.8 5.0 translate 0.0 0.0 -2.5
B = cylinder orange scale 0.8 0.8 5.0 translate 0.0 0.0 -2.5 rotate 0.0
1.0 0.0 90.0
Tree = (A+B)
Tree = (A.B)
Tree = (A-B)
Tree = ((X1.X2)-(A+B))
```

Figure A.3: Sample CsgDemo Input File

A.2 CSG Tools

Several separate command line tools have been developed for the purpose of research, testing and demonstration. These are typically used to supply a complete CSG model to the CsgDemo program as follows:

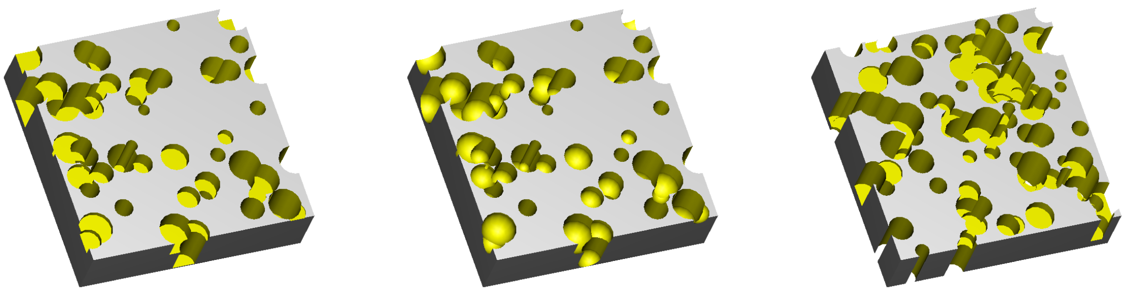
```
tool | csgdemo -
```

A.2.1 Three Axis Milling

The program `3axis` produces a randomised three axis CSG model consisting of a rectangular block and subtracted endmill or ballnose shaped holes.

```
3axis type seed n
```

- `type` is `0` for endmill (square) shaped holes, or `1` for ballnose (spherical) shaped holes.
- `seed` is the random number seed for varying the position and size of subtracted holes.
- `n` is the number of subtracted holes.



3axis 0 251274 50

3axis 1 251274 50

3axis 0 251074 100

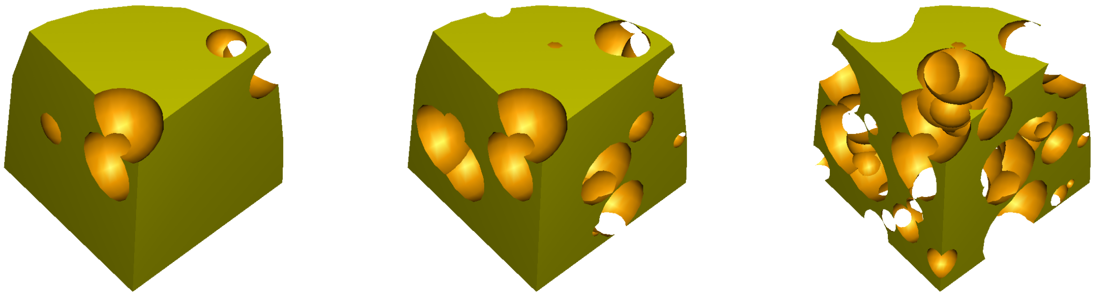
Figure A.4: Three Axis Milling Examples

A.2.2 Swiss Cheese

The program `cheese` produces a randomised Swiss cheese CSG model consisting of a CSG rounded block and subtracted holes.

```
cheese type radius seed n
```

- `type` is `0` for an intersected cheese block, or `1` for subtracted. As the SCS algorithm can intersect more efficiently than it can subtract, `0` is the recommended mode of operation.
- `radius` is the maximum radius of subtracted holes. The Swiss cheese model is approximately 1 unit in width, height and depth. A maximum hole radius of between 0.15 and 0.30 is recommended.
- `seed` is the random number seed for varying the position and size of subtracted holes.
- `n` is the number of subtracted holes.



cheese 1 0.2 251274 10 cheese 1 0.2 251274 25 cheese 1 0.2 251274 100

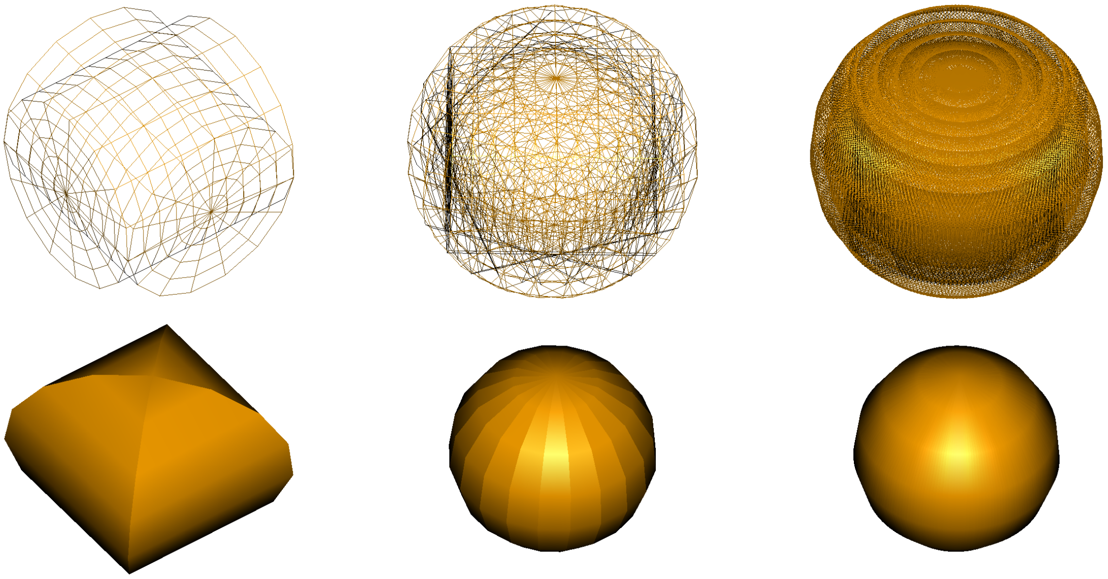
Figure A.5: Swiss Cheese Examples

A.2.3 Cylinders

The program `cylinders` produces a CSG model consisting of n intersected cylinders. This program is primarily intended for benchmarking the performance of the SCS convex intersection algorithm.

```
cylinders n
```

- `n` is the number of cylinders.



cylinders 2 cylinders 10 cylinders 100

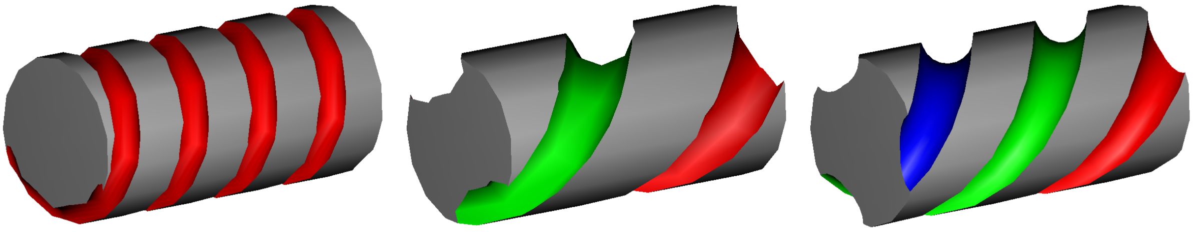
Figure A.6: Cylinders Examples

A.2.4 Helix

The program helix produces a CSG model composed of helical volumes subtracted from a solid cylinder. These are intended to resemble the arrangement of swept volumes subtracted from a metal blank in CNC tool manufacturing. The helical volumes are composed of convex components for the SCS rendering algorithm. For the Goldfeather algorithm, concave helical volumes are produced.

```
helix n convex stacks slices radius freq
```

- n is the number of helical volumes.
- convex is 1 for convex volumes, 0 for concave.
- slices is the number of subdivisions along the helical path.
- stacks is the number of subdivisions around the circle swept along the helical path.
- freq is the number of revolutions of the helical path.



helix 1 1 40 20 0.2 5 helix 2 1 8 8 0.4 1 helix 3 1 20 20 0.3 1

Figure A.7: Helix Examples

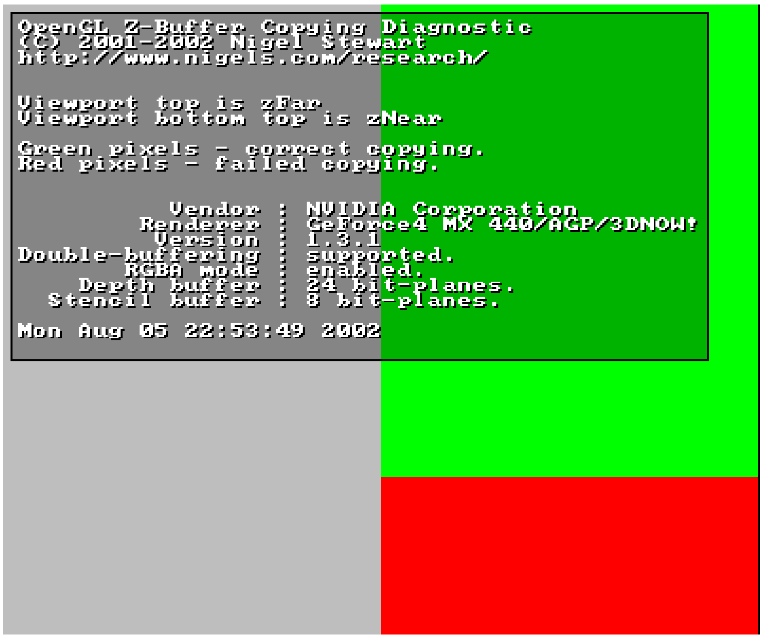
A.3 OpenGL Diagnostic Tools

A.3.1 OpenGL Buffer Copying Benchmark

The program `glcopysp` times the OpenGL `glCopyPixels()` call for copying depth, stencil of colour buffer information within an OpenGL window. Depth buffer copying can be a substantial bottleneck in CSG rendering algorithms. The performance characteristics of several platforms is included in Table A.3. A more detailed examination of OpenGL buffer copying performance is in Section 7.2.

Graphics Hardware	colour	depth	stencil
		(10^6 pixel/s)	
SGI IMPACT/2/2/4	17	22	34
SGI IRL/S/1/16/4 Onyx Irix 6.4	30	19	42
GeForce3	320	0.5	320
GeForce4 MX 440/AGP/3DNOW!	300	0.16	400
GeForce4 Ti 4200	470	0.5	420
GeForceFX 5600	480	0.5	320
GeForce 6150 LE/PCI/SSE2/3DNOW!	80	0.47	182
GeForce 7600 GS/PCI/SSE2/3DNOW!	390	0.35	184

Table A.3: OpenGL Buffer Copying Performance



```
OpenGL Z-Buffer Copying Diagnostic
(C) 2001-2002 Nigel Stewart
http://www.nigels.com/research/


Viewport top is zFar
Viewport bottom top is zNear

Green pixels - correct copying.
Red pixels - failed copying.


          Vendor : NVIDIA Corporation
        Renderer : GeForce4 MX 440/AGP/3DNOW!
         Version : 1.3.1
Double-buffering : supported.
       RGBA mode : enabled.
    Depth buffer : 24 bit-planes.
  Stencil buffer : 8 bit-planes.

Mon Aug 05 22:53:49 2002
```

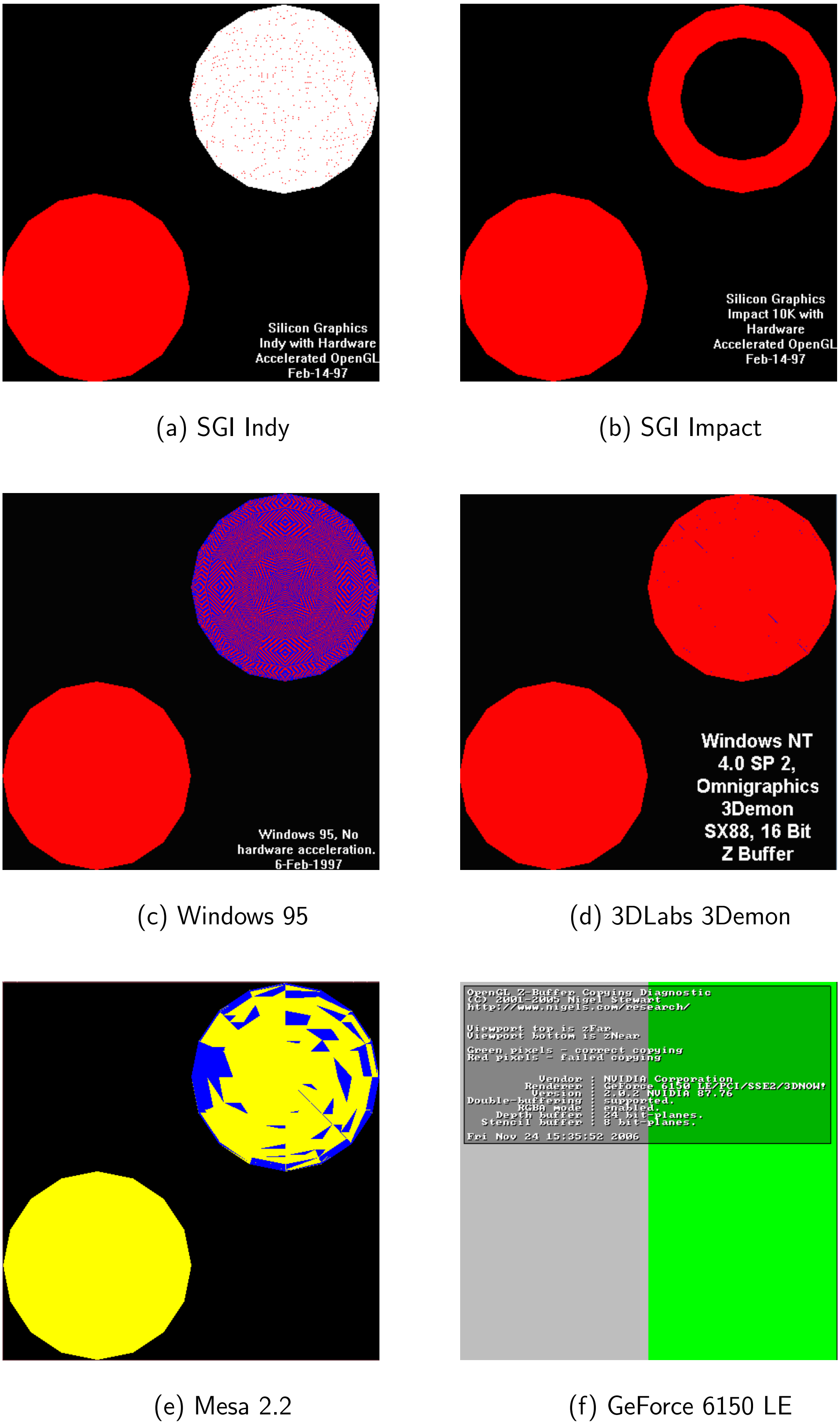
Figure A.8: Output of `glzcopy` Diagnostic for Nvidia GeForce4 MX

A.3.2 OpenGL Depth Buffer Copying Diagnostic

The program `glzcopy` demonstrates a potential problem with the accuracy of depth buffer copying within an OpenGL window. The window is divided into two viewports: left and right. The left viewport is covered by a quadrilateral from z_{near} at the bottom to z_{far} at the top. The z-buffer is then copied from the left viewport to the right viewport. The same quadrilateral is drawn to the right viewport with a z-equal depth test. Where the depth test passes, green pixels are drawn. Where the depth test fails, red pixels are drawn.

The result of this diagnostic varies for different OpenGL implementations. Figure A.8 illustrates the result on Nvidia GeForce4 MX hardware. Depth buffer precision is lost for the front quarter of the frustum. Therefore for this OpenGL implementation it is important to restrict use of the frustum to the reliable region of z-buffer copying.

Figure A.9 illustrates results collected for a variety of platforms. In previous versions of the program a sphere was drawn rather than a quadrilateral.



```
OpenGL Z-Buffer Copying Diagnostic
(C) 2001-2005 Nigel Stewart
http://www.nigels.com/research/

Viewport top is zFar
Viewport bottom is zNear

Green pixels - correct copying
Red pixels - failed copying

          Vendor : NVIDIA Corporation
        Renderer : GeForce 6150 LE/PCI/SSE2/3DNOW!
         Version : 2.0.2 NVIDIA 87.76
Double-buffering : supported.
       RGBA mode : enabled.
    Depth buffer : 24 bit-planes.
  Stencil buffer : 8 bit-planes.

Fri Nov 24 15:35:52 2006
```

(a) SGI Indy

(b) SGI Impact

(c) Windows 95

(d) 3DLabs 3Demon

(e) Mesa 2.2

(f) GeForce 6150 LE

Figure A.9: `glzcopy` Results for Various Platforms

Appendix B

Source Code

As we know, there are known knowns.
There are things we know we know.
We also know there are known unknowns.
That is to say, we know there are some things we do not know.
But there are also unknown unknowns, the ones we don't know we don't know.
— Donald H. Rumsfeld

B.1 Goldfeather CSG Rendering

B.1.1 Goldfeather CSG Rendering Algorithm

The routine `csgRenderGoldfeather` implements the Goldfeather CSG rendering algorithm, as described in Section 3.5.1. The routine takes as input a CSG `sumOfProducts`, runtime configuration `options`, and an `information` object for collecting runtime statistics. The stencil and depth buffers are assumed to be cleared before the routine is called.

The routine makes use of a `CsgDepthBufferHelper` to simulate a second OpenGL depth buffer. This second depth buffer can `read` and `write` to and from the OpenGL depth buffer.

The routine makes use of `csgRenderLightShadeZBuffer` for converting the final depth buffer into a lit and shaded RGB colour buffer.

The routine makes use of `parityIntersect`, `paritySubtract` and `parityApply` subroutines for managing stencil parity bit-masks, the stencil buffer and clipping the depth buffer.

Nvidia graphics hardware requires some special handling due to inconsistency between the fragments generated with lighting turned on or off. Lighting can be disabled on other graphics hardware, and enabled for the final pass that generates the RGB image of the CSG model. For Nvidia hardware, a depth test of `GL_EQUAL` is only reliable if lighting is enabled or disabled for all passes of the algorithm. For other hardware, disabling lighting calculations for intermediate passes may result in a performance improvement. The option `options.nvidiaHardware` should be enabled whenever Nvidia hardware is detected by the application.

Upon completion the CSG rendered result is stored in the OpenGL colour and depth buffers. The OpenGL state is restored, except for the stencil buffer which is undefined. The `GLERROR` macro is used to trap any OpenGL errors that occur before or during the execution of the routine.

```
void csgRenderGoldfeather
(
    const CsgSumOfProducts &sumOfProducts,
    const CsgOptions       &options,
          CsgInfo          &information
)
{
    Timer timer;            // Track elapsed time
    GLERROR                 // Check for OpenGL errors

    glPushAttrib(GL_ENABLE_BIT | GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT |
                 GL_POLYGON_BIT | GL_VIEWPORT_BIT);

        // Simulated second OpenGL depth buffer
        CsgDepthBufferHelper zBuffer2(options.zBufferCopy);

        // Massage OpenGL transformation matricies, if necessary
        if (options.msDriver)
            massageOpenGLMatrices();

    // Query OpenGL for number of stencil bit-planes,
    // usually eight

    GLint stencilBits = 0;
    glGetIntegerv(GL_STENCIL_BITS,&stencilBits);
    information.stencilBits = GLuint(stencilBits);

    // Do not draw into RGBA, or do expensive
    // things like lighting or normalisation

    glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);
    glDisable(GL_NORMALIZE);

    // Nvidia hardware generates different fragments depending
    // on the lighting mode.  The depth will need to match
    // when csgRenderLightShadeZBuffer is called.

    if (options.nvidiaHardware)
        glEnable(GL_LIGHTING);
    else
        glDisable(GL_LIGHTING);

    // Process each CSG product, in turn

    bool useSecondZbuffer = false;
    GLuint p;

    // For products of size greater than one, each layer
    // of each primitive needs to be clipped individually

    glDepthFunc(GL_ALWAYS);
    glDepthMask(GL_TRUE);

    for (p=0; p<sumOfProducts.size(); p++)
        if (sumOfProducts[p].size()>1)
        {
            const CsgProduct &product = sumOfProducts[p];

            // Non-trivial products are clipped,
            // each layer of each primitive at a time.

            for (GLuint i=0; i<product.size(); i++)
            {
                // Determine depth complexity of the shape,
                // each layer will be clipped in turn

                glEnable(GL_CULL_FACE);
                glCullFace(GL_BACK);
                const GLuint maxK = countSurfaces(*product.shape(i),COUNT_SURFACES_ALL);

                for (GLuint k=0; k<maxK; k++)
                {
                    information.layerClips++;

                    // Copy OpenGL depth buffer to the 2nd buffer,
                    // for merging back in later

                    if (useSecondZbuffer)
                    {
                        zBuffer2.read();
                        information.depthReads++;
                    }

                    // In the convex case, only one layer to be
                    // processed for this primitive.

                    if (maxK==1)
                    {
                        glDisable(GL_STENCIL_TEST);

                        // Draw front or back surface into z-buffer
                        if (product.positive(i))
                            product[i].drawFront();
                        else
                            product[i].drawBack();
                    }
                    else
                    {
                        glClear(GL_STENCIL_BUFFER_BIT);
                        information.stencilClears++;

                        glEnable(GL_STENCIL_TEST);
                        glStencilOp(GL_INCR,GL_INCR,GL_INCR);
                        glStencilFunc(GL_EQUAL,k,~0);
                        glStencilMask(~0);

                        if (product.positive(i))
                            product[i].drawFront();
                        else
                            product[i].drawBack();

                        glClear(GL_STENCIL_BUFFER_BIT);
                        information.stencilClears++;
                    }

                    // Clip depth buffer against other primitives
                    // in the CSG product.

                    GLuint bit  = 0;
                    GLuint mask = 0;

                    for (GLuint j=0; j<product.size(); j++)
                    {
                            // Only clip against other shapes

                            if (j!=i)
                            {
                                // The parity test is inverted depending on
                                // whether we're intersecting or subtracting.

                                if (product.positive(j))
                                    parityIntersect(*product.shape(j),bit,mask);
                                else
                                    paritySubtract(*product.shape(j),bit,mask);

                                bit++;
                            }

                            // Apply parity mask as soon as stencil bit planes
                            // have been exhausted, or finished with clipping

                            if ( (bit==GLuint(stencilBits) || j==product.size()-1) && bit )
                            {
                                // Pixels that don't match the parity mask
                                // are reset to zFar

                                parityApply(mask,~0);
                                glClear(GL_STENCIL_BUFFER_BIT);
                                information.stencilClears++;

                                mask = 0;
                                bit = 0;
                            }
                        }

                        // Merge with 2nd z-buffer, if available

                        if (useSecondZbuffer)
                        {
                            zBuffer2.write(true);
                            information.depthWrites++;
                        }

                        // Preserve 1st z-buffer next time
                        useSecondZbuffer = true;
                    }
                }
            }

        // (OPTIMISATION)
        //
        // For products of size one, no clipping is necessary
        // Simply draw front surfaces with z-less test

        glDisable(GL_STENCIL_TEST);
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LESS);
        glDepthMask(GL_TRUE);

        for (p=0; p<sumOfProducts.size(); p++)
            if (sumOfProducts[p].size()==1)
                sumOfProducts[p].front().drawFront();

        // Now that depth buffer contains CSG result,
        // the lit and shaded RGB result can be drawn.

        csgRenderLightShadeZBuffer(sumOfProducts,options);

    glPopAttrib();                             // Restore OpenGL state
    GLERROR                                    // Check for errors
    glFinish();                                // Wait for OpenGL to finish

    information.drawTime += timer.clock(); // Collect elapsed time for analysis
}
```

B.1.2 Parity Testing

The routine `parityTest` implements depth buffer parity testing. The routine takes as input a `shape`, the stencil `bit` to be used to store pixel parity, and the OpenGL `depthFunction` to be used.

The routine is used via the `parityIntersect` and `paritySubtract` routines which serve to manage the stencil parity mask used by the `parityApply` routine described in the next section.

The parity of the depth buffer is even outside the volume of the shape, and odd when inside. In the case of intersection, pixels of even parity are subsequently clipped. In the case of subtraction, pixels of odd parity are subsequently clipped. Clipping passes are minimised by forming a parity mask which is applied once all the stencil bit-planes are in use.

Upon completion the depth buffer parity is stored in the specified stencil bitplane, leaving the other stencil bitplanes intact. The OpenGL state is restored and the depth and colour buffers left intact. The `GLERROR` macro is used to trap any OpenGL errors that occur before or during the execution of the routine.

```
void parityIntersect(const GltShape &shape,const GLuint bit,GLuint &mask,const GLenum depthFunction)
{
    parityTest(shape,bit,depthFunction);
    mask |= (1<<bit);  // Set parity mask bit
}
```

```
void paritySubtract(const GltShape &shape,const GLuint bit,GLuint &mask,const GLenum depthFunction)
{
    parityTest(shape,bit,depthFunction);
    mask &= ~(1<<bit);  // Clear parity mask bit
}
```

```
void parityTest(const GltShape &shape,const GLuint bit,const GLenum depthFunction)
{
    GLERROR

    const GLuint mask = 1<<bit;

    glPushAttrib(GL_ENABLE_BIT        | GL_DEPTH_BUFFER_BIT   |
                 GL_COLOR_BUFFER_BIT  | GL_STENCIL_BUFFER_BIT);

        glEnable(GL_STENCIL_TEST);                  // Configure stencil test to toggle
        glStencilFunc(GL_ALWAYS,~0,~0);             // the mask bit for each pixel for
        glStencilOp(GL_KEEP,GL_KEEP,GL_INVERT);     // each fragment.

        glEnable(GL_DEPTH_TEST);                    // Configure the depth test
        glDepthFunc(depthFunction);                 // (usually GL_LESS), and
        glDisable(GL_CULL_FACE);                    // disable back-face culling

        glStencilMask(mask);
        glDepthMask(GL_FALSE);
        glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);

        shape.draw();

    glPopAttrib();

    GLERROR
}
```

B.1.3 Depth Buffer Parity Clipping

The routine `parityApply` implements depth buffer parity clipping. The routine takes as input a stencil `parityMask` that specifies the desired parity of each bit-plane in the stencil buffer, and a `stencilMask` that specifies which stencil planes should be considered.

The depth buffer is updated according to the parity and stencil masks and the stencil is left intact. Upon completion the OpenGL state is restored. The `GLERROR` macro is used to trap any OpenGL errors that occur before or during the execution of the routine.

```
void parityApply(const GLuint parityMask,const GLuint stencilMask)
{
    GLERROR

    glPushAttrib(GL_ENABLE_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

        glEnable(GL_STENCIL_TEST);                                  // Use stencil testing to
        glStencilFunc(GL_NOTEQUAL,parityMask,stencilMask);          // restrict fragments to pixels
        glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);                       // not satisfying the parity mask

        glEnable(GL_DEPTH_TEST);                                    // Enable depth testing
        glDepthFunc(GL_ALWAYS);                                     // Depth test always passes
        glDisable(GL_CULL_FACE);                                    // Disable face culling

        glStencilMask(0);                                           // Disable stencil updates
        glDepthMask(GL_TRUE);                                       // Enable depth updates

        drawZfar();                                                 // Draw viewport size polygon

    glPopAttrib();

    GLERROR
}
```

B.1.4 Z-Buffer Shade and Light

The routine `csgRenderLightShadeZBuffer` implements shading and lighting of the depth buffer, as described in Section 4.2.5. The routine takes as input a CSG `sumOfProducts` and runtime configuration `options`. Shading and lighting of the OpenGL depth buffer surface is achieved by redrawing the shapes in the CSG tree with an OpenGL depth test of `GL_EQUAL`.

Use of face culling is optional, and is configured as part of the `options` parameter. With culling enabled, fewer triangles are rasterised, resulting in fewer fragments, and a possible improvement to performance. With culling disabled, the algorithm correctly handles geometry without consistent face-winding, but may be less efficient.

Upon completion the lit and shaded result is stored in the OpenGL colour buffer. The OpenGL state is restored, and the stencil and depth buffers remain unchanged.

```
void csgRenderLightShadeZBuffer
(
    const CsgSumOfProducts &sumOfProducts,
    const CsgOptions       &options
)
{
    GLERROR         // Check for OpenGL errors

    glPushAttrib(GL_ENABLE_BIT | GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT | GL_POLYGON_BIT);

        // Match z-buffer with primitive surfaces

        glDisable(GL_STENCIL_TEST);                     // No stencil testing

        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_EQUAL);                          // z-equal depth test
        glDepthMask(GL_FALSE);                          // No updates to depth buffer

        glEnable(GL_LIGHTING);
        glEnable(GL_NORMALIZE);
        glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);   // Draw into colour buffer

        // Face culling is normally used, but can be
        // disabled at some expense to performance

        if (options.cullFaces)
            glEnable(GL_CULL_FACE);
        else
            glDisable(GL_CULL_FACE);

        // Draw front facing surfaces of intersected objects

        glCullFace(GL_BACK);

        for (GLuint i=0; i<sumOfProducts.size(); i++)
        {
            const CsgProduct &product = sumOfProducts[i];

            for (GLuint j=0; j<product.size(); j++)
                if (product.positive(j))
                    product.shape(j)->draw();
        }

        // Draw back facing surfaces of subtracted objects

        glCullFace(GL_FRONT);

        for (GLuint i=0; i<sumOfProducts.size(); i++)
        {
            const CsgProduct &product = sumOfProducts[i];

            for (GLuint j=0; j<product.size(); j++)
                if (!product.positive(j))
                    product.shape(j)->draw();
        }

    glPopAttrib();  // Restore OpenGL state

    GLERROR         // Check for errors
}
```

B.2 Goldfeather CSG Rendering Variants

B.2.1 Layered Goldfeather CSG Rendering Algorithm

The routine `csgRenderGoldfeatherLayered` implements a variation [87] of the Goldfeather CSG rendering algorithm, as described in Section 3.5.2. The same remarks in relation to the Goldfeather implementation in Appendix B.1.1 apply here.

```
void csgRenderGoldfeatherLayered
(
    const CsgSumOfProducts &sumOfProducts,
    const CsgOptions       &options,
          CsgInfo          &information
)
{
    Timer timer;          // Track elapsed time

    GLERROR               // Check for OpenGL errors

    glPushAttrib(GL_ENABLE_BIT | GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT |
                 GL_POLYGON_BIT | GL_VIEWPORT_BIT);

        // Simulated second OpenGL depth buffer
        CsgDepthBufferHelper zBuffer2(options.zBufferCopy);
        bool useSecondZbuffer = false;

        // Massage OpenGL transformation matricies, if necessary
        if (options.msDriver)
            massageOpenGLMatrices();

        // Query OpenGL for number of stencil bit-planes
        GLint stencilSize = 0;
        glGetIntegerv(GL_STENCIL_BITS,&stencilSize);

        // Do not draw into RGBA, or do expensive
        // things like lighting or normalisation

        glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);
        glDisable(GL_NORMALIZE);

        // NVIDIA hardware generates different fragments depending on the lighting mode

        if (options.nvidiaHardware)
            glEnable(GL_LIGHTING);
        else
            glDisable(GL_LIGHTING);

        // Process each CSG product, in turn

        GLuint p;

        // For products of size greater than one, each layer
        // of each primitive needs to be clipped individually

        glDepthFunc(GL_ALWAYS);
        glDepthMask(GL_TRUE);

        for (p=0; p<sumOfProducts.size(); p++)
            if (sumOfProducts[p].size()>1)
            {
                const CsgProduct &product = sumOfProducts[p];

                // Here's the variation on Goldfeather.
                // Start with front facing surfaces of non-inverted primitives.
                // Extract the 0 to kth layer and clip against _all_ primitives.

                // Seperate elements into front (intersected) and back (subtracted).
                // Front or back face culling is used, depending on whether the element is subtracted.

                GltShapes front,back;
                product.split(front,back);

                const GLint frontK = countSurfaces(front,COUNT_SURFACES_ALL);
                const GLint backK  = countSurfaces(back ,COUNT_SURFACES_ALL);
                const GLint kMax   = frontK + backK;

                // Collect statistics
                information.depthComplexityFront += frontK;
                information.depthComplexityBack  += backK;
                information.depthComplexity      += kMax;

                assert(kMax<(1<<stencilSize));

                for (GLint k=0; k<kMax; k++)
                {
                    // Copy OpenGL depth buffer to the 2nd buffer, for merging back in later

                    if (useSecondZbuffer)
                    {
                        zBuffer2.read();
                        information.depthReads++;
                    }

                    // Stencil needs to be cleared for layer counting
                    glClear(GL_STENCIL_BUFFER_BIT);
                    information.stencilClears++;

                    // Draw k'th layer into z-buffer

                    glDepthFunc(GL_ALWAYS);
                    glDepthMask(GL_TRUE);
                    glEnable(GL_CULL_FACE);

                    glEnable(GL_STENCIL_TEST);
                    glStencilOp(GL_INCR,GL_INCR,GL_INCR);
                    glStencilMask(~0);

                    if (k<frontK)
                    {
                        glCullFace(GL_BACK);
                        glStencilFunc(GL_EQUAL,k,~0);
                        front.draw();
                    }
                    else
                    {
                        glCullFace(GL_FRONT);
                        glStencilFunc(GL_EQUAL,k-frontK,~0);
                        back.draw();
                    }

                    // Clip depth buffer against all primitives in the CSG product.

                    GLint bit = 0;
                    GLuint mask = 0;

                    // Stencil needs to be cleared for parity clipping
                    glClear(GL_STENCIL_BUFFER_BIT);
                    information.stencilClears++;

                    for (GLuint j=0; j<product.size(); j++)
                    {
                        // The parity test is inverted depending on
                        // whether we're intersecting or subtracting.

                        if (product.positive(j))
                            parityIntersect(*product.shape(j),bit,mask,GL_LEQUAL);
                        else
                            paritySubtract(*product.shape(j),bit,mask,GL_LEQUAL);

                        bit++;

                        // Apply parity mask as soon as stencil bit planes
                        // have been exhausted, or finished with clipping

                        if ( (bit==stencilSize || j==product.size()-1) && bit )
                        {
                            // Pixels that don't match the parity mask are reset to zFar

                            parityApply(mask,~0);
                            glClear(GL_STENCIL_BUFFER_BIT);
                            information.stencilClears++;

                            mask = 0;
                            bit = 0;
                        }
                    }

                    // Merge with 2nd z-buffer, if available

                    if (useSecondZbuffer)
                    {
                        zBuffer2.write(true);
                        information.depthWrites++;
                    }

                    // Preserve 1st z-buffer next time
                    useSecondZbuffer = true;
                }
            }

        // Optimisation
        //
        // For products of size one, no clipping is necessary
        // Simply draw front surfaces with z-less test

        glDisable(GL_STENCIL_TEST);
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LESS);
        glDepthMask(GL_TRUE);

        for (p=0; p<sumOfProducts.size(); p++)
            if (sumOfProducts[p].size()==1)
                sumOfProducts[p].front().drawFront();

        // Now that depth buffer contains CSG result,
        // the lit and shaded RGB result can be drawn.

        csgRenderLightShadeZBuffer(sumOfProducts,options);

    glPopAttrib();                         // Restore OpenGL state
    GLERROR                                // Check for errors
    glFinish();                            // Wait for OpenGL to finish

    information.drawTime += timer.clock(); // Collect elapsed time for analysis
}
```

B.2.2 Improved Layered Goldfeather CSG Rendering Algorithm

The routine csgRenderGoldfeatherLayered implements a variation [29] of the Goldfeather CSG rendering algorithm, as described in Section 3.5.3. The same remarks in relation to the Goldfeather implementation in Appendix B.1.1 apply here.

The routine makes use of a GltFrameBufferStencilUbyte object for copying stencil buffers into memory. A C++ std::list [51] container is used for storing all the necessary stencil buffers in memory.

The routine partitions the stencil buffer bit-planes into lower and higher portions. The lower bits are used for counting during layer extraction and for storing the result of intermediate parity tests. The higher bits are used for storing per-layer masks for combining layers in the final phase. stencilBits is the total size of the OpenGL stencil buffer. lowerBits is the number of bits used for layer counting and parity testing, lowerMask is used as a bit-mask. The parityBit accesses one of the lower bits and is masked with parityMask. During parity testing, parityApply subroutine is applied to reset the depth of clipped pixels to z_{far}. The layerBit is the current high bit for flagging un-clipped pixels, and is masked with layerMask. The layer bit-planes are formed by detecting pixels closer than z_{far}. Layer bit-planes are transferred into main memory once the capacity of the stencil buffer has been reached.

The number of lower bit-planes is minimised by choosing only enough bits to count layers of front or back surfaces. This way, the number of bit-planes utilised for layer masks is maximised, and as a consequence the copying of stencil buffers between the frame buffer and main memory is minimised.

While we regard this ‘improved’ Goldfeather algorithm to be superior to the previous two variants, we have found it to be the most intricate and complicated to implement. This is mainly due to the way that stencil bit-planes are interpreted differently at various stages of the algorithm.

A potential refinement to this algorithm is to completely eliminate the need for depth-buffer copying, making use of stencil buffers for all layers of all products and merging all of these as a final step. This may complicate the implementation even further, since each product may partition the stencil buffer differently. However, we think the performance advantage may be significant, depending on the relative speed of copying depth buffer and stencil buffer data on a particular hardware platform.

```
void csgRenderGoldfeatherLayeredImproved
(
    const CsgSumOfProducts &sumOfProducts,
    const CsgOptions       &options,
          CsgInfo          &information
)
{
    Timer timer;            // Track elapsed time
    GLERROR                 // Check for OpenGL errors

    glPushAttrib(GL_ENABLE_BIT | GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT |
                 GL_POLYGON_BIT | GL_VIEWPORT_BIT);

        // Simulated second OpenGL depth buffer
        CsgDepthBufferHelper zBuffer2(options.zBufferCopy);
        bool useSecondZbuffer = false;

        // Massage OpenGL transformation matricies, if necessary
        if (options.msDriver)
            massageOpenGLMatrices();

        // Query OpenGL for number of stencil bit-planes
        GLint stencilBits = 0;
        glGetIntegerv(GL_STENCIL_BITS,&stencilBits);

        // Process each CSG product, in turn

        // For products of size greater than one, each layer
        // of each primitive needs to be clipped individually

        for (GLuint p=0; p<sumOfProducts.size(); p++)
            if (sumOfProducts[p].size()>1)
            {
                // Do not draw into RGBA, or do expensive
                // things like lighting or normalisation

                glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);
                glDisable(GL_NORMALIZE);

                // NVIDIA hardware generates different fragments depending
                // on the lighting mode

                if (options.nvidiaHardware)
                    glEnable(GL_LIGHTING);
                else
                    glDisable(GL_LIGHTING);

                glDepthFunc(GL_ALWAYS);
                glDepthMask(GL_TRUE);

                // Copy OpenGL depth buffer to the 2nd buffer,
                // for merging back in later

                if (useSecondZbuffer)
                {
                    zBuffer2.read();
                    information.depthReads++;
                }

                //

                const CsgProduct &product = sumOfProducts[p];

                // Seperate elements into front (intersected) and
                // back (subtracted).  Front or back face culling is
                // used, depending on whether the element is subtracted.

                GltShapes front,back;
                product.split(front,back);

                // Find the depth complexity of front-facing and
                // back-facing surfaces seperately.

                const GLuint frontK = countSurfaces(front,COUNT_SURFACES_ALL);
                const GLuint backK  = countSurfaces(back ,COUNT_SURFACES_ALL);

                //

                const GLuint maxK   = MAX(frontK,backK);
                const GLuint totalK = frontK+backK;

                // Determine the number of bits required for
                // layer counting.

                GLuint lowerBits = 0;
                for (GLuint kTmp = maxK; kTmp>0; lowerBits++,kTmp>>=1);

                // Mask the lowest lowerBits bits in the stencil buffer
                // for layer counting and parity testing.  The other
                // stencil bits are reserved for layer parity masks.

                const GLuint lowerMask = (1<<lowerBits)-1;

                // Collect statistics
                information.depthComplexityFront += frontK;
                information.depthComplexityBack  += backK;
                information.depthComplexity      += totalK;
                information.stencilBufferKBits    = lowerBits;

                // Check that we have enough stencil bits to
                // do layer counting upto maxK

                assert(maxK <(1<<stencilBits));
assert(maxK <(1<<lowerBits));

// layerBit is the current stencil bit for layer parity mask

GLuint layerBit = lowerBits;

// Layer parity masks that don't fit into
// stencil planes must be copied into memory

list<GltFrameBufferStencilUbyte *> stencilBuffers;

// Find the stencil mask for each layer

for (GLuint k=0; k<totalK; k++)
{
    // Clear layer counting planes of
    // stencil buffer in preparation
    // for layer counting

    glStencilMask(lowerMask);
    glClear(GL_STENCIL_BUFFER_BIT);
    information.stencilClears++;

    // Setup depth testing to draw all
    // fragments that pass the stencil test.

    glDepthFunc(GL_ALWAYS);
    glDepthMask(GL_TRUE);
    glEnable(GL_CULL_FACE);

    // Draw k'th layer into z-buffer

    glEnable(GL_STENCIL_TEST);
    glStencilOp(GL_INCR,GL_INCR,GL_INCR);
    glStencilMask(lowerMask);

    if (k<frontK)
    {
        glCullFace(GL_BACK);
        glStencilFunc(GL_EQUAL,k,lowerMask);
        front.draw();
    }
    else
    {
        glCullFace(GL_FRONT);
        glStencilFunc(GL_EQUAL,k-frontK,lowerMask);
        back.draw();
    }

    //
    // Parity test layer against all primitives
    //

    // parityBit is the current stencil parity bit
    // parityMask is the combined stencil parity mask

    // Parity information is using the same bitplanes
    // as layer counting.

    GLuint parityBit = 0;
    GLuint parityMask = 0;

    // Clear lower bit-planes in preparation for parity testing

    glStencilMask(lowerMask);
    glClear(GL_STENCIL_BUFFER_BIT);
    information.stencilClears++;

    // Do parity testing

    for (GLuint j=0; j<product.size(); j++)
    {
        if (product.positive(j))
            parityIntersect(*product.shape(j),parityBit,parityMask,GL_LEQUAL);
        else
            paritySubtract (*product.shape(j),parityBit,parityMask,GL_LEQUAL);

        // Use the next available stencil bit
        parityBit++;

        // If we've run out of stencil bits, or
        // we're finished with parity testing, it's
        // time to apply the parity mask

        if ( (parityBit==lowerBits || j==product.size()-1) && parityBit )
        {
            // Reset pixels that fail parity test

            parityApply(parityMask,lowerMask);

            // Clear parity testing stencil planes

            glStencilMask(lowerMask);
            glClear(GL_STENCIL_BUFFER_BIT);
            information.stencilClears++;

            // Reset mask and current bit

            parityMask = 0;
            parityBit = 0;
        }
    }

```

```
            // Now that parity clipping is complete, create a binary mask
            // for pixels with z<zFar.  We'll use this flag later to
            // merge all the clipped surfaces into the final image.

            const GLuint layerMask = (1<<layerBit);

            glStencilMask(layerMask);
            glStencilFunc(GL_ALWAYS,layerMask,layerMask);
            glStencilOp(GL_ZERO,GL_ZERO,GL_REPLACE);

            glDepthMask(GL_FALSE);
            glDepthFunc(GL_GREATER);
            glDisable(GL_CULL_FACE);

            drawZfar();

            // Select the next stencil buffer bit for layer
            // mask

            layerBit++;

            // If we've run out of stencil buffer bits, it's time
            // to read the stencil buffer into main memory

            if (layerBit==GLuint(stencilBits) && k+1<totalK)
            {
                layerBit = lowerBits;

                stencilBuffers.push_back(new GltFrameBufferStencilUbyte());
                information.stencilReads++;
            }
        }

        // Replace saved zBuffer if possible, otherwise
        // clear depth buffer to z-far

        if (useSecondZbuffer)
        {
            zBuffer2.write(false);
            information.depthWrites++;
        }
        else
        {
            // Preserve OpenGL z-buffer next time
            useSecondZbuffer = true;

            glDepthMask(GL_TRUE);
            glClear(GL_DEPTH_BUFFER_BIT);
            information.depthClears++;
        }

        // Use the stencil masks to mask each layer and merge the final
        // result.  Work in reverse order, since there may be masks in
        // the stencil buffer before we start copying stencil buffers
        // from memory.

        for (k=totalK-1; k<totalK; k--)
        {
            // Select the previous stencil buffer bit for layer
            // mask

            layerBit--;

            // If we've run out of stencil buffer bits, it's time
            // to copy the next memory buffer into the stencil

            if (layerBit<lowerBits)
            {
                layerBit = stencilBits-1;

                stencilBuffers.back()->write();
                delete stencilBuffers.back();
                stencilBuffers.pop_back();
                information.stencilWrites++;
            }

            // Clear layer counting planes of
            // stencil buffer in preparation
            // for layer counting

            glStencilMask(lowerMask);
            glClear(GL_STENCIL_BUFFER_BIT);
            information.stencilClears++;

            // Draw k'th layer into z-buffer

            glDepthFunc(GL_LESS);
            glDepthMask(GL_TRUE);
            glEnable(GL_CULL_FACE);

            glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
            glEnable(GL_NORMALIZE);
            glEnable(GL_LIGHTING);

            const GLuint layerMask = (1<<layerBit);

            glEnable(GL_STENCIL_TEST);
            glStencilOp(GL_INCR,GL_INCR,GL_INCR);
            glStencilMask(lowerMask);

            if (k<frontK)
            {
                glCullFace(GL_BACK);
```

```
                        glStencilFunc(GL_EQUAL,k|layerMask,lowerMask|layerMask);
                        front.draw();
                    }
                    else
                    {
                        glCullFace(GL_FRONT);
                        glStencilFunc(GL_EQUAL,(k-frontK)|layerMask,lowerMask|layerMask);
                        back.draw();
                    }
                }

                // Hopefully, no memory leaks!
                assert(stencilBuffers.size()==0);
            }

        // For CSG products of size one, simply
        // draw with z-less depth test

        glDepthFunc(GL_LESS);
        glDepthMask(GL_TRUE);

        glEnable(GL_DEPTH_TEST);
        glDisable(GL_STENCIL_TEST);

        glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
        glEnable(GL_NORMALIZE);
        glEnable(GL_LIGHTING);

        glEnable(GL_CULL_FACE);
        glCullFace(GL_BACK);

        for (p=0; p<sumOfProducts.size(); p++)
            if (sumOfProducts[p].size()==1)
                sumOfProducts[p].shape(0)->draw();

    glPopAttrib();                         // Restore OpenGL state
    GLERROR                                // Check for errors

    glFinish();                            // Wait for OpenGL to finish
    information.drawTime += timer.clock(); // Collect elapsed time for analysis
}
```

B.2.3 Depth Complexity Sampling

The routine `countSurfaces` implements stencil based depth complexity sampling as described in Section 3.4.1. The routine takes a shape as input. The depth complexity of each pixel is stored in the stencil buffer as output. The stencil buffer is cleared as necessary by the routine, and the depth buffer is ignored. The `GLERROR` macro is used to trap any OpenGL errors that occur before or during the execution of the routine.

Additional programming is required to copy the stencil buffer into memory and determine the overall depth complexity by examining all of the stencil values for the maximum.

```
void countSurfaces(const GltShape &shape)
{
    GLERROR

    glPushAttrib(GL_ENABLE_BIT | GL_DEPTH_BUFFER_BIT |
                 GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

        // Ensure stencil buffer is fully zero

        glClearStencil(0);
        glStencilMask(~0);
        glClear(GL_STENCIL_BUFFER_BIT);

        // Disable z buffer test & update

        glDepthMask(GL_FALSE);
        glDepthFunc(GL_ALWAYS);

        // Use stencil test to count surfaces

        glEnable(GL_STENCIL_TEST);
        glStencilFunc(GL_ALWAYS,0,~0);
        glStencilOp(GL_INCR,GL_INCR,GL_INCR);

        // Disable updates to colour buffer

        glDisable(GL_LIGHTING);
        glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);

        // Draw all objects
        shape.draw();

    glPopAttrib();

    GLERROR
}
```

B.3 SCS CSG Rendering

B.3.1 Convex Intersection

The routine `csgRenderConvexIntersection` implements convex intersection of objects in the OpenGL depth buffer, as described in Section 4.2.1. The routine takes as input a CSG `product`, runtime configuration `options`, and an `information` object for collecting runtime statistics. Intersection is performed in the OpenGL depth buffer and also makes use of the OpenGL stencil buffer. The depth and stencil buffers are cleared as necessary by the routine, and therefore do not need to be initialised by the calling routine.

Nvidia graphics hardware requires some special handling due to inconsistency between the fragments generated with lighting turned on or off. Lighting can be disabled on other graphics hardware, and enabled for the final pass that generates the RGB image of the CSG model. For Nvidia hardware, a depth test of `GL_EQUAL` is only reliable if lighting is enabled or disabled for all passes of the algorithm. For other hardware, disabling lighting calculations for intermediate passes may result in a performance improvement. The option `options.nvidiaHardware` should be enabled whenever Nvidia hardware is detected by the application.

In the special case that there is only one intersected object in the CSG product, the front facing polygons are simply drawn to the depth buffer with a `GL_ALWAYS` depth test. In this situation it is also unnecessary to clear the stencil buffer or use stencil testing.

The routine makes use of a function `drawZfar()` that draws a viewport covering polygon at the far clipping plane of the viewing frustum. Fragments of this polygon are stencil tested to determine which pixels should be set to z_{far}, and which should be retained.

Upon completion the intersected result is stored in the OpenGL depth buffer. The OpenGL state is restored, and the stencil buffer contains only zeros. The `GLERROR` macro is used to trap any OpenGL errors that occur before or during the execution of the routine.

```
void csgRenderConvexIntersection
(
    const CsgProduct &product,           // CSG Product
    const CsgOptions &options,           // Runtime Options
          CsgInfo    &information        // Runtime Info
)
{
    GLERROR                              // Check for OpenGL errors

    CsgProduct intersect,subtract;       // Partition CSG product into
    product.split(intersect,subtract);   // intersected and subtracted

    // Preserve OpenGL state

    glPushAttrib(GL_ENABLE_BIT | GL_POLYGON_BIT | GL_DEPTH_BUFFER_BIT |
                 GL_STENCIL_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

        // Enable writes to depth and stencil buffers only

        glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);
        glDepthMask(GL_TRUE);
        glStencilMask(~0);

        // Disable stencil testing and normalisation

        glDisable(GL_STENCIL_TEST);
        glDisable(GL_NORMALIZE);

        // Disable lighting, unless using NVIDIA hardware

        if (options.nvidiaHardware)
            glEnable(GL_LIGHTING);
        else
            glDisable(GL_LIGHTING);

        // Draw the furthest front facing surface into z-buffer.

        const int n = intersect.size();

        glClearStencil(0);
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_CULL_FACE);
        glCullFace(GL_BACK);

        // Clear depth buffer, as necessary

        if (n>1)
        {
            glClearDepth(0.0);
            glDepthFunc(GL_GREATER);
            glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

            information.depthClears++;
            information.stencilClears++;
        }
        else
        {
            glDepthFunc(GL_ALWAYS);
            glClear(GL_STENCIL_BUFFER_BIT);

            information.stencilClears++;
        }

        // Draw all intersected objects

        int i;
        for (i=0; i<n; i++)
            intersect.shape(i)->draw();

        // If there is more than one intersected object,
        // perform image-space clipping

        if (n>1)
        {
            // Count the number of back-facing surfaces behind each pixel.

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

            for (i=0; i<n; i++)
                intersect.shape(i)->draw();

            // Reset the z-buffer for pixels where stencil != n
            // Also, reset stencil to zero

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

            // Draw viewport covering polygon at zFar
            drawZfar();

            information.intersections += n;
        }

    glPopAttrib();                       // Restore OpenGL state
    GLERROR                              // Check for OpenGL errors
}
```

B.3.2 Convex Subtraction

The routine csgRenderConvexSubtraction implements sequenced SCS convex subtraction of objects from the OpenGL depth buffer, as described in Section 4.2.2. The routine takes as input a CSG product, a subtraction sequence, runtime configuration options, and an information object for collecting runtime statistics. Subtraction is performed on the OpenGL depth buffer which would normally contain the front faces of the intersected objects in the CSG product.

Use of face culling is optional, and is configured as part of the options parameter. With culling enabled, fewer triangles are rasterised, resulting in fewer fragments, and a possible improvement to performance. With culling disabled, the algorithm correctly handles geometry without consistent face-winding, but may be less efficient.

Nvidia graphics hardware requires some special handling due to inconsistency between the fragments generated with lighting turned on or off. Lighting can be disabled on other graphics hardware, and enabled for the final pass that generates the RGB image of the CSG model. For Nvidia hardware, a depth test of GL_EQUAL is only reliable if lighting is enabled or disabled for all passes of the algorithm. For other hardware, disabling lighting calculations for intermediate passes may result in a performance improvement. The option options.nvidiaHardware should be enabled whenever Nvidia hardware is detected by the application.

This implementation detects the number of available stencil bits and aims to minimise the number of stencil clear operations. It is assumed that the stencil buffer is cleared to zero before the csgRenderConvexSubtraction routine is called. The variable stencilCode is used to mark pixels in the first pass that may need updating in the second pass. This variable is incremented until it exceeds the size of the stencil buffer. In this case the stencil buffer is cleared and stencilCode is reset to 1. An 8-bit stencil buffer needs to cleared after each 255th entry in the subtraction sequence.

Several runtime statistics are gathered by the routine and stored in the information object. These statistics include the total number of subtraction operations, the number of stencil buffer clears, and the number of stencil buffer bits.

Upon completion the subtracted result is stored in the OpenGL depth buffer. The OpenGL state is restored, with the exception of the stencil buffer which contains arbitrary information left over from subtraction operations. The GLERROR macro is used to trap any OpenGL errors that occur before or during the execution of the routine.

```
void csgRenderConvexSubtraction
(
    const CsgProduct              &product,       // CSG Product
    const CsgSubtractionSequence  &sequence,      // Subtraction Sequence
    const CsgOptions              &options,       // Runtime Options
          CsgInfo                 &information    // Runtime Info
)
{
    GLERROR                                       // Check for OpenGL errors

    glPushAttrib(GL_ENABLE_BIT | GL_POLYGON_BIT | GL_DEPTH_BUFFER_BIT |
                 GL_STENCIL_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

        // Enable writes to depth and stencil buffers only

        glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);
        glDepthMask(GL_TRUE);
        glStencilMask(~0);

        // Enable stencil and depth testing, disable normalisation

        glEnable(GL_STENCIL_TEST);
        glEnable(GL_DEPTH_TEST);
        glDisable(GL_NORMALIZE);

        // Culling mode is optional (perhaps faster)

        if (options.cullFaces)
            glEnable(GL_CULL_FACE);
        else
            glDisable(GL_CULL_FACE);

        // Disable lighting, unless using NVIDIA hardware

        if (options.nvidiaHardware)
            glEnable(GL_LIGHTING);
        else
            glDisable(GL_LIGHTING);

        // Determine the number of stencil buffer bits

        GLint stencilBits = 0;
        glGetIntegerv(GL_STENCIL_BITS,&stencilBits);
        assert(stencilBits>0);

        information.stencilBits = GLuint(stencilBits);

        const GLuint stencilMask = (1<<stencilBits)-1;
              GLuint stencilCode = 0;

        // Subtract each object in the sequence...

        for (GLuint i=0; i<sequence.size(); i++)
        {
            // Clear stencil buffer when necessary

            if (++stencilCode>stencilMask)
            {
                stencilCode = 1;
                glClear(GL_STENCIL_BUFFER_BIT);
                information.stencilClears++;
            }

            // Get the i'th entry in the sequence

            const GLuint j = sequence[i];
            assert(!product.positive(j));

            const GltShapePtr &shape = product.shape(j);
            assert(shape.get());

            // Set stencil code for front facing surfaces
            // closer or equal to the z-buffer

            glCullFace(GL_BACK);
            glDepthFunc(GL_LEQUAL);
            glDepthMask(GL_FALSE);
            glStencilFunc(GL_ALWAYS,stencilCode,stencilMask);
            glStencilOp(GL_KEEP,GL_KEEP,GL_REPLACE);

            shape->draw();

            // Replace z-buffer with back facing surface
            // iff greater than z-buffer and stencil code matches

            glCullFace(GL_FRONT);
            glDepthFunc(GL_GEQUAL);
            glDepthMask(GL_TRUE);
            glStencilFunc(GL_EQUAL,stencilCode,stencilMask);
            glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

            shape->draw();
        }

    glPopAttrib();                                  // Restore OpenGL state

    GLERROR;                                        // Check for OpenGL errors

    information.subtractions += sequence.size();    // Collect subtraction statistics
}
```

B.3.3 Z-Buffer Clip

The routine `csgRenderClipZBuffer` implements OpenGL depth buffer clipping, as described in Section 4.2.3. The routine takes as input a CSG `product`. Clipping is performed on the OpenGL depth buffer which would normally contain the result of image-space intersection and subtraction of objects in the product. The `GLERROR` macro is used to trap any OpenGL errors that occur before or during the execution of the routine.

```
void csgRenderClipZBuffer(const CsgProduct &prod)
{
    GLERROR                                // Check for OpenGL errors

    glPushAttrib(GL_ENABLE_BIT | GL_STENCIL_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // Clear the stencil buffer
        glClear(GL_STENCIL_BUFFER_BIT);

        // Cull front facing polygons
        glCullFace(GL_FRONT);
        glEnable(GL_CULL_FACE);

        // Configure z-less depth test with
        // no updates to depth buffer
        glDepthFunc(GL_LESS);
        glDepthMask(GL_FALSE);

        // Configure stencil test to set pixels
        // to one which pass the depth test
        glEnable(GL_STENCIL_TEST);
        glStencilFunc(GL_ALWAYS,1,~0);
        glStencilOp(GL_KEEP,GL_KEEP,GL_REPLACE);

        // Draw all intersected objects in the CSG product
        for (GLuint i=0; i<prod.size(); i++)
            if (prod.positive(i))
                prod.shape(i)->draw();

        // Configure OpenGL for second pass, using
        // a stencil test to reset pixels with stencil
        // equal to one
        glDisable(GL_CULL_FACE);
        glDepthFunc(GL_ALWAYS);
        glDepthMask(GL_TRUE);
        glStencilFunc(GL_EQUAL,1,0xffff);
        glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

        // Draw viewport covering polygon at zFar
        drawZfar();

    glPopAttrib();                         // Restore OpenGL state
    GLERROR                                // Check for OpenGL errors
}
```

B.4 Subtraction Sequence Generation

The *Sequenced Convex Subtraction* (SCS) algorithm as described in Chapters 4 and 6 utilises *Permutation Embedding Sequences* (PESs) as subtraction sequences for CSG rendering. This section presents a C++ implementation of PES generation as used by SCS.

B.4.1 Adleman Subtraction Sequence

The routine `gscsSequenceAdleman` encodes a Permutation Embedding Sequence (PES) for a given number of objects n, where $n \geq 3$. The Adleman method [14] is described in Section 5.3.3. The routine outputs a subtraction sequence of length $n^2 - 2n + 4$ as a C++ `std::vector` of unsigned integers:

```
typedef std::vector<unsigned int> CsgSubtractionSequence;
```

```
CsgSubtractionSequence gscsSequenceAdleman(const unsigned int n)
{
    // n must be at least 3
    assert(n>=3);
    if (n<3)
        return CsgSubtractionSequence();

    // Storage for generated sequence.

    CsgSubtractionSequence tmp;
    tmp.reserve(n*n-2*n+4);

    // The sequence begins with the n'th symbol
    tmp.push_back(n-1);

    // Repeat the string abcd..abcd..
    // With the n'th symbol inserted after
    // the i'th occurance of the symbol n-i
    // where 1 <= i <= n-2

    const unsigned int size = n*n-3*n+4;

    unsigned int k=n-2;
    for (unsigned int i=0; i<size; i++)
    {
        const unsigned int j = i%(n-1);
        tmp.push_back(j);

        // Insert the n'th symbol

        if (j==k && k>0)
        {
            tmp.push_back(n-1);
            k--;
        }
    }

    // The sequence ends with the n'th symbol
    tmp.push_back(n-1);

    return tmp;
}
```

B.4.2 Galbiati Subtraction Sequence

The routine `gscsSequenceGalbiati` encodes a Permutation Embedding Sequence (PES) for a given number of objects n, where $n \geq 4$. The Galbiati method [35] is described in Section 5.3.4. The routine outputs a subtraction sequence of length $n^2 - 2n + 4$ as a C++ `std::vector` of unsigned integers:

```
typedef std::vector<unsigned int> CsgSubtractionSequence;
```

```
CsgSubtractionSequence gscsSequenceGalbiati(const unsigned int n)
{
     // n must be at least 4

    assert(n>=4);
    if (n<4)
        return CsgSubtractionSequence();

    // Storage for generated sequence.

    CsgSubtractionSequence tmp;
    tmp.reserve(n*n-2*n+4);

    // Galbiati uses four arbitrary elements: 1,2,3,4
    // We use zero as base: 0,1,2,3

    tmp.push_back(1);
    tmp.push_back(0);
    tmp.push_back(2);

    // Append each segment

    const unsigned int V[3] = { 1,2,3 };
    unsigned int v = 2;

    for (unsigned int i=0; i<n-1; i++)
    {
        // Append U

        for (unsigned int j=4; j<n; j++)
            tmp.push_back(j);

        // Append next element in V
        tmp.push_back(V[v++%3]);
        tmp.push_back(0);
        tmp.push_back(V[v++%3]);
    }

    return tmp;
}
```

B.4.3 Savage Subtraction Sequence

The routine `gscsSequenceSavage` encodes a Permutation Embedding Sequence (PES) for a given number of objects n and depth complexity k, where $3 \leq k \leq n$. The Savage method [83] is described in Section 5.3.5. The routine outputs a subtraction sequence of length $kn - 2k + 4$ as a C++ `std::vector` of unsigned integers. In the case that $k = n$ the length is $n^2 - 2n + 4$.

```
typedef std::vector<unsigned int> CsgSubtractionSequence;
```

```
CsgSubtractionSequence gscsSequenceSavage(const unsigned int n, const unsigned int k)
{
    // n and k must be at least 3

    assert(n>=3 && k>=3);
    if (n<3 || k<3)
        return CsgSubtractionSequence();

    // Storage for generated sequence.

    CsgSubtractionSequence tmp;
    tmp.reserve(k*n-2*k+4);

    // The n elements are partitioned into two
    // sets: A = ab..x_{k-1}, B = x_{k}..x_{n}

    // The sequence begins with B
    unsigned int m;
    for (m=k-1; m<n; m++)
        tmp.push_back(m);

    // Repeat the string AA..
    // With the B symbol inserted after
    // the i'th occurance of the symbol n-i
    // where 1 <= i <= k-2

    const unsigned int size = k*k-3*k+4;

    unsigned int p=k-2;
    for (unsigned int i=0; i<size; i++)
    {
        const unsigned int j = i%(k-1);
        tmp.push_back(j);

        // Insert B

        if (j==p && p>0)
        {
            for (m=k-1; m<n; m++)
                tmp.push_back(m);
            p--;
        }
    }

    // The sequence ends with B
    for (m=k-1; m<n; m++)
        tmp.push_back(m);

    return tmp;
}
```

B.4.4 Combined Subtraction Sequence Generation

The routine `gscsSequence` encodes a Permutation Embedding Sequence (PES) for a given number of objects n and depth complexity k. A table of "best known" subtraction sequences is used for $0 < n < 8$ and $k \leq n$. Otherwise the Galbiati or Savage methods are used, if possible. If necessary, the simple method described in Section 5.3.2 is used to produce a sequence of length $n^2 - n + 1$.

This routine is utilised by the Sequenced Convex Subtraction (SCS) CSG rendering algorithm to determine the subtraction sequence for each frame. If depth complexity sampling is used, k will tend to be less than n. If overlap graph processing is used, PES for cyclic subgraphs are combined into an overall subtraction sequence as described in Section 6.3.

The use of a lookup table is based on the possibility of determining PES of shorter length than those produced by other methods such as Galbiati or Savage. Sequence encoding as implemented here has not been observed to be a performance issue, except for depth complexity sampling associated with overlap graph processing. The subtraction sequences appearing in the lookup table correspond to Table 5.7 and Table 5.8 in Section 5.4.8 — PES proven computationally to be shortest length.

```
CsgSubtractionSequence gscsSequence(const unsigned int n, const unsigned int k)
{
    // Best known subtraction sequences

    static const uint32 bestSequences = 8;
    static const char *bestSequence[bestSequences][bestSequences] =
    {
        { NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL },  // n=0
        { NULL, "a",  NULL, NULL, NULL, NULL, NULL, NULL },  // n=1
        { NULL, "ab", "aba",NULL, NULL, NULL, NULL, NULL },  // n=2

        // n=3
        {
            NULL,
            "abc",
            "abcab",
            "abcabac",
            NULL, NULL, NULL, NULL
        },

        // n=4
        {
            NULL,
            "abcd",
            "abcdabc",
            "abcdabcabd",
            "abcdabcadbac",
            NULL, NULL, NULL
        },

        // n=5
        {
            NULL,
            "abcde",
            "abcdeabcd",
            "abcdeabcdabce",
            "abcdeabcdabecabd",
            "abcdeabcdaebcadbcea",
            NULL, NULL
        },

        // n=6
        {
            NULL,
            "abcdef",
            "abcdefabcde",
            "abcdefabcdefabcd",
            NULL, NULL, NULL, NULL
        },

        // n=7
        {
            NULL,
            "abcdefg",
            "abcdefgabcdef",
            NULL, NULL, NULL, NULL, NULL
        }
    };

    // Check sanity of parameters.

    assert(k<=n);
    assert(n>=0);
    assert(k>=0);

    // Use the best-known sequence if available

    if (n<bestSequences && k<bestSequences && bestSequence[n][k])
    {
        CsgSubtractionSequence tmp;
        for (const char *i = bestSequence[n][k]; *i!='\0'; i++)
            tmp.push_back(*i-'a');
        return tmp;
    }

    // Use Galbiati if possible

    if (n>=4 && k==n)
        return gscsSequenceGalbiati(n);

    // Use Savage, if possible

    if (n>=3 && k>=3)
        return gscsSequenceSavage(n,k);

    // Use n^2-n+1 sequence if necessary

    CsgSubtractionSequence tmp;
    tmp.reserve(n*n-n+1);

    //

    for (unsigned int i=0; i<k; i++)
        for (unsigned int j=0; j<n; j++)
            if (i==0 || j>0)
                tmp.push_back( (i%2)==0 ? j : n-1-j );

    return tmp;
}
```

B.5 Shortest Length Subtraction Sequences

This section presents key components of our C++ implementation of *Normalised No Repeat* (NNR) search as detailed in Section 5.4.2.

B.5.1 NNR Size

The routine `NNRsize` calculates the number of NNR sequences of a particular length. Two rows of storage are allocated dynamically and utilised to evaluate the recursive expression detailed in Section 5.4.4. An arbitrary precision `Integer` class is used for intermediate and output values, since the number of possible NNR sequences can potentially exceed $2^{32}-1$. The algorithm requires $O(nl)$ time and $O(n)$ storage.

$$|\Omega(n,l)| = (n-1) \times |\Omega(n,l-1)| + |\Omega(n-1,l-1)|$$

$$n = 1,\ l = 1\ \rightarrow |\Omega| = 1$$

$$n > 1,\ l = 1\ \rightarrow |\Omega| = 0$$

$$n = 1,\ l > 1\ \rightarrow |\Omega| = 0$$

```
Integer NNRsize(const uint32 n,const uint32 length)
{
    if (length<n || n<1 || length<1)
        return 0;

    // Allocate storage for two rows of length n
    Integer *buffer = new Integer[2*n];
    assert(buffer);

    // Table is initialised to 1,0,0,...
    Integer *row = buffer;
    row[0] = 1;
    for (uint32 k=1; k<n; k++)
        row[k] = 0;

    // Next row follows current row in memory
    Integer *next = buffer+n;

    // Calculate subsequent rows of table iteratively,
    // based on values of previous row.

    for (uint32 i=1; i<length; i++)
    {
        next[0] = 0;

        // Apply recursive definition
        for (uint32 j=1; j<n; j++)
            next[j] = row[j]*j + row[j-1];

        // Update row pointer
        swap(row,next);
    }

    // NNR size is n'th element of last row.
    Integer size = row[n-1];

    // Free resources
    delete [] buffer;

    return size;
}
```

B.5.2 NNR Partitions

The routine `NNRpartitions` calculates the number of NNR partitions of a particular length, as discussed in Section 5.4.5. An arbitrary precision `Integer` class is used for intermediate and output values, since the number of possible NNR partitions can potentially exceed $2^{32} - 1$.

$$|\Gamma| = \frac{(l-1)!}{(n-1)!(l-n)!} \qquad n \geq 1,\ l \geq 1,\ l \geq n$$

```
Integer
NNRpartitions(const uint32 n,const uint32 length)
{
    Integer x(1);
    Integer y(1);
    Integer z(1);

    if (length>1)   x = fac(length-1);
    if (n>1)        y = fac(n-1);
    if (length-n>1) z = fac(length-n);

    return x/(y*z);
}
```

B.5.3 NNR Partition Size

The routine `NNRpartitionSize` calculates the number of NNR sequences for a particular partition p, as discussed in Section 5.4.5. An arbitrary precision `Integer` class is used for intermediate and output values, since the number of possible NNR partitions can potentially exceed $2^{32} - 1$.

$$|\gamma| = 1^{|s_2|-1} \times\ 2^{|s_3|-1} \times\ 3^{|s_4|-1} \times\ \cdots\ \times\ (n-1)^{|s_n|-1}$$

```
Integer
NNRpartitionSize(const std::vector<uint32> &p)
{
    PesInteger size = 1;

    for (uint32 i=1; i<p.size(); i++)
        size *= pow(Integer(i),Integer(p[i]-1));

    return size;
}
```

B.5.4 NNR Partition Sequence

The routine `NNRsequence` determines the i^{th} NNR sequence for a particular partition p, as discussed in Section 5.4.6. An arbitrary precision `Integer` class is used for intermediate and output values, since the number of possible NNR sequences can potentially exceed $2^{32} - 1$.

Following the NNR sequence structure, the first entry in each segment is always fixed. The following entries in each segment are chosen according to `idx` and can not repeat the previous entry. Integer modulus division is used to determine the choice of entries based on the desired unique sequence in the partition. Using this approach adjacent sequences have little lexicographic similarity due to the least significant bits of `i` being used for the leftmost variable elements of the sequence.

```
Sequence
NNRsequence(const std::vector<uint32> &p,const Integer &i)
{
    assert(p.size()>0);

    // idx is used to choose elements in each segment
    Integer idx(i);

    // Output sequence
    Sequence seq(p.size());

    // For each segment...
    for (uint32 j=0; j<p.size(); j++)
    {
        // Each segment begins with n'th element
        seq.push_back(j);

        // First segment can only have length of one
        if (j>0)
        {
            // For each subsequent entry in the segment
            for (uint32 k=1; k<p[j]; k++)
            {
                // Make a choice based on idx and segment
                uint32 c = idx%j;

                // No repeats, based on previous entry
                seq.push_back((seq.back()+1+c)%(j+1));

                // Adjust idx
                idx /= j;
            }
        }
    }

    return seq;
}
```

References

[1] OpenGL 3D and 2D graphics API. `http://www.opengl.org`, Feb 2008.

[2] Khronos Group. `http://www.khronos.org`, Feb 2008.

[3] SGI OpenGL. `http://www.sgi.com/products/software/opengl`, Feb 2008.

[4] The Mesa 3D graphics library. `http://www.mesa3d.org`, Feb 2008.

[5] Seti@home. `http://setiathome.berkeley.edu`, Feb 2008.

[6] Folding@home. `http://folding.stanford.edu`, Feb 2008.

[7] Graphviz graph drawing software. `http://www.research.att.com/sw/tools/graphviz`, Feb 2008.

[8] SCS CSG rendering algorithm. `http://www.nigels.com/research`, Feb 2008.

[9] Intel page on Moore's Law. `http://www.intel.com/technology/mooreslaw`, Feb 2008.

[10] General-purpose computation using graphics hardware. `http://www.gpgpu.org`, Feb 2008.

[11] BRL-CAD cross-platform solid modelling system. `http://brlcad.org`, Feb 2008.

[12] P. C. van Oorschot A. J. Mezenes and S. A. Vanstone. *Handbook of applied cryptography.* CRC Press, 1997.

[13] B. Adams and P. Dutré. Interactive boolean operations on surfel-bounded solids. *ACM Trans. Graph.*, 22(3):651–656, 2003.

[14] L. Adleman. Short permutation strings. *Discrete Mathematics*, 10:197–200, 1974.

[15] K. Akeley, P. Brown, C. Frazier, J. Leech, and M. Segal. *The OpenGL Graphics System: A Specification*, 2004.

[16] P. Atherton. A scan-line hidden surface removal procedure for constructive solid geometry. *Computer Graphics (Proc Siggraph)*, 17(3):73–82, Jul 1983.

[17] J. Ayala, P. Brunet, R. Juan, and I. Navazo. Object representation by means of nonminimal division quadtrees and octrees. *ACM Trans. on Graphics*, 4(1):41–59, Jan 1985.

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

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

[20] K. Bouatouch, M. O. Madani, T. Priol, and B. Arnaldi. A new algorithm for space tracing using a csg model. *Proc. Eurographics '87*, pages 65–78, 1987.

[21] K. Paterson C. Mitchell. Perfect factors from cyclic codes and interleaving. *Siam J. Discrete Math.*, 11(2):241–264, 1998.

[22] S. Cameron. Efficient intersection tests for objects defined constructively. *Int. J. Rob. Res.*, 8(1):3–25, 1989.

[23] S. Cameron and J. Rossignac. Relationship between S-bounds and active zones in constructive solid geometry. *Proc. of Theory and Practice of Geometric Modeling*, pages 369–382, Oct 1988.

[24] J. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[25] J. Cohen, M. Lin, D. Manocha, and K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scaled environments. *Proceedings of ACM Int. 3D Graphics Conference*, pages 189–196, 1995.

[26] F. Crow. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248, New York, NY, USA, 1977. ACM.

[27] R. Diestal. *Graph Theory.* Sprinter-Verlag, 2000.

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

[29] G. Erhart and R. Tobler. General purpose Z-buffer CSG rendering with consumer level hardware. *VRVis Technical Report 2000-03*, 2000.

[30] R. Erra, N. Lygeros, and N. Stewart. On minimal strings containing the elements of S_n by decimation. *Discrete Mathematics & Theoretical Computer Science*, AA:165–176, 2001.

[31] C. Everitt and M. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering, Mar 2002.

[32] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. Pixelflow: The realization. *Proc. 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, pages 3–13, Aug 1997.

[33] L. H. Figueiredo F. Romeiro, L. Velho. Hardware-assisted rendering of CSG models. *XIX Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI '06)*, pages 139–146, Oct 2006.

[34] H. Fredricksen. A survey of full length nonlinear shift register cycle algorithms. *SIAM Review*, 24:195–221, April 1982.

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

[36] 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.

[37] 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.

[38] S. Golomb. *Shift register sequences.* Holden-Day, San Fransisco, 1967.

[39] I. J. Good. Normal recurring decimals. *Journal of the London Mathematical Society*, 21:167–169, 1946.

[40] M. Goodrich. Applying parallel processing techniques to classification problems in constructive solid geometry. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 118–128, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[41] A. Gottlieb. Cracking the secret of the baltimore hilton. *Technology Review*, Feb 1980. cited in [34].

[42] S. Guha, S. Krishnan, K. Munagala, and S. Venkatasubramanian. Application of the two-sided depth test to CSG rendering. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 177–180, New York, NY, USA, 2003. ACM Press.

[43] J. Hable and J. Rossignac. Blister: GPU-based rendering of boolean combinations of free-form triangulated shapes. *ACM Trans. Graph.*, 24(3):1024–1031, 2005.

[44] G. Herres. Real time constructive solid geometry rendering using 3D texture mapping. *J. Comput. Small Coll.*, 19(5):333–335, 2004.

[45] G. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-1(2):145–153, Apr 1979.

[46] J. Rossignac J. Hable. CST: Constructive solid trimming for rendering BReps and CSG. *GVU Tech Report GIT-GVU-06-16*, pages 1–10, Sep 2006.

[47] F. Jansen. A csg list priority hidden surface algorithm. *Proceedings Eurographics '85*, pages 51–62, 1985.

[48] F. Jansen. CSG hidden surface algorithms for vlsi hardware systems. *Advances in Graphics Hardware I*, pages 75–82, 1987.

[49] F. Jansen. Depth-order point classification techniques for CSG display algorithms. *ACM Transactions on Graphics*, 10(1):40–70, Jan 1991.

[50] F. Jansen and R. Sutherland. Display of solid models with a multi-processor system. *Proceedings Eurographics'87*, pages 377–387, 1987.

[51] N. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley, 1999.

[52] F. Kirsch and J. Döllner. Rendering techniques for hardware-accelerated image-based CSG. *WSCG 2004*, 2(12):221–228, 2004.

[53] F. Kirsch and J. Döllner. OpenCSG: A library for image-based CSG rendering. *2005 USENIX Annual Technical Conference*, pages 129–140, 2005.

[54] D. E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, 1976.

[55] D. E. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, 3rd edition.* Addison-Wesley, 1997.

[56] P. J. Koutas and T. C. Hu. Shortest string containing all permutations. *Discrete Mathematics*, 11:125–132, 1975.

[57] E. Lawler. *Combinatorial Optimization : Networks and Matroids.* Dover Publications, 1976.

[58] D. Liao and S. Fang. Fast volumetric csg modeling using standard graphics system. In *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, pages 204–211, New York, NY, USA, 2002. ACM.

[59] M. Mano. *Computer Engineering: Hardware Design.* Prentice Hall, 1988.

[60] Flye-Sainte Marie. Solution to problem number 58. *L'Intermédiaire des Mathématiciens*, 1:107–110, 1894.

[61] M. H. Martin. A problem in arrangements. *Bulletin of the American Mathematical Society*, 40:859–864, 1934.

[62] M. Mazzetti and L. Ciminiera. Computing CSG tree boundaries as algebraic expressions. In *SMA '93: Proceedings on the second ACM symposium on Solid Modeling and Applications*, pages 155–162, New York, NY, USA, 1993. ACM.

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

[64] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. *SIGGRAPH '92*, pages 231–240, 1992.

[65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[66] T. Myer and I. Sutherland. On the design of display processors. *Commun. ACM*, 11(6):410–414, June 1968.

[67] M. Nienhaus, F. Kirsch, and J. Döllner. Illustrating design and spatial assembly of interactive CSG. In *Afrigraph '06: Proceedings of the 4^{th} international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 91–98, New York, NY, USA, 2006. ACM Press.

[68] N. Okino, Y. Kakazu, and M. Morimoto. Extended depth-buffer algorithms for hidden-surface visualization. *IEEE CG&A*, 4(5):79–88, May 1984.

[69] Modern Machine Shop Online. Better production software provides the edge for tool grinding, Sep 2001. `http://www.mmsonline.com/articles/0801bp3.html`, Feb 2008.

[70] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Eurographics 2005 - State of the Art Reports*, pages 21–51, 2005.

[71] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[72] A. Rappoport. The n-dimensional extended convex differences tree (ecdt) for representing polyhedra. In *SMA '91: Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications*, pages 139–147, New York, NY, USA, 1991. ACM.

[73] 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.

[74] A. Requicha. Representations for rigid solids: Theory, methods, and systems. *Computing Surveys*, 12(4):437–464, Dec 1980.

[75] 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.

[76] J. Rossignac. Optimized blist form (OBF). *GVU Tech Report GIT-GVU-07-10.*

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

[78] J. Rossignac. Blist: A boolean list formulation of CSG trees. *GVU Tech Report GIT-GVU-99-04*, pages 1–9, Oct 1998.

[79] J. Rossignac and A. Requicha. Depth-buffering display techniques for constructive solid geometry. *IEEE CG&A*, 6(9):29–39, Sep 1986.

[80] J. Rossignac and H. Voelcker. Active zones in CSG for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms. *ACM Trans. Graph.*, 8(1):51–87, 1989.

[81] S. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, New York, NY, USA, 1980. ACM.

[82] H. Sato, M. Ishii, K. Sato, M. Ikesaka, H. Ishihata, M. Kakimoto, K. Hirota, and K. Inoue. Fast image generation of construcitve solid geometry using a cellular array processor. In *Proceedings of the* 12^{th} *annual conference on Computer graphics and interactive techniques*, pages 95–102. ACM Press, 1985.

[83] C. Savage. Short strings containing all k-element permutations. *Discrete Mathematics*, 42:281–285, 1982.

[84] C. E. Shannon. *A Symbolic Analysis of Relay and Switching Circuits.* Massachusetts Institute of Technology, Dept. of Electrical Engineering, 1940.

[85] S. Stein. *Mathematics, The Man-Made Universe.* Freeman, San Fransisco, 1976.

[86] I. Stewart. *Game, Set and Math.* Penguin Mathematics, 1989. published initially in *Scientific American.*

[87] 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.

[88] 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.

[89] N. Stewart, G. Leach, and S. John. Linear-time CSG rendering of intersected convex objects. *The* 10^{th} *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2002 - WSCG 2002*, II:437–444, Feb 2002.

[90] N. Stewart, G. Leach, and S. John. Improved CSG rendering using overlap graph subtraction sequences. *International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia - GRAPHITE 2003*, pages 47–53, Feb 2003.

[91] W. Stürzlinger. Bounding volume construction using point clouds. *Spring Conference on Computer Graphics '96*, pages 239—-246, June 1996.

[92] I. Sutherland, R. Sproull, and R. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, 1974.

[93] 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*, II:379–386, Feb 2001.

[94] R. B. Tilove. A null-object detection algorithm for constructive solid geometry. *Commun. ACM*, 27(7):684–694, 1984.

[95] I. Tuomi. The lives and death of moore's law. *First Monday*, 7(11), November 2002.

[96] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984.

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