MODELLING SOLIDS IN MOTION

Stephen Alan Cameron

Ph.D. Thesis

University of Edinburgh

1984
Abstract

This thesis is concerned with the problem of producing and using a computer model that describes a set of solid objects in motion; in particular, I consider the efficient solution to the clash detection problem as encountered in the field of robotics.

The r-set model of three-dimensional solid objects is taken and extended to four dimensions to give a mathematical basis for the modelling of objects in motion, and I show that such models can be constructed with relative ease when working in a constructive solid geometry (CSG) framework. The clash detection problem is the problem of deciding whether a collision will occur between a set of moving objects, and I show that this is equivalent to deciding whether any pair of four-dimensional models intersect.

Three basic methods are considered for performing clash detection using a computer. The first tackles the problem by repeatedly testing for static interference between objects over many values of time. The second considers the four-dimensional models of the objects in motion directly, and performs an interference test between these. The third method creates a new set of three-dimensional models by sweeping the original objects through space, and then performs a test for interference between these.

The first method has been implemented in my CSG based geometric modelling system, robmod, and includes several refinements based on the calculation of the minimum distance between objects. The second method has also been implemented for a certain class of allowable motions, and both robmod and the implementations are described herein. All three methods rely on being able to tell whether a pair of multidimensional sets intersect; this problem has been examined in some detail and I describe a new, efficient solution that I have developed.
Acknowledgements

Although this thesis is my own work my thanks are due to the many people who have helped and encouraged me over the last few years. My thanks go to my supervisors, Robin Popplestone and Pat Ambler, for their time, their guidance and their occasional cajolery; I also wish to express my particular thanks to my friends, John Hallam and Roy Featherstone, for their painstaking reading of earlier drafts of this thesis and for the multitude of suggestions that they have made regarding it and the work reported within it. John is also responsible for a simplification to a proof in Appendix D, and Roy provided much of the code for the original version of the robmod parser.

My thanks also extend to the many friends and colleagues, in Edinburgh, Leeds, Bath and elsewhere, who have helped to bring together in my mind the various ideas presented in this thesis over many hours of discussion; and last, but by no means least, I would like to thank Fran for her moral support whilst this thesis was in preparation.

This work was supported by a Science and Engineering Research Council studentship.

Declaration

This thesis has been composed by myself and describes my own work.
# Contents

List of Sections 5  
Abbreviations and Symbols 9  

Chapters  
1. Modelling Solids in Motion 10  
2. Adding Motions to Models 20  
3. The Clash Detection Problem 37  
4. Basic Concepts and Techniques 51  
5. robmod 69  
6. S-Bounds 77  
7. Null Object Detection 93  
8. Statistical Analysis of Algorithms 113  
9. Multiple Interference Detection 132  
10. Four Dimensional Modelling and Sweeping 150  
11. Clash Detection Systems 165  
12. Conclusions and Further Research 177  

Appendices  
A. Set Theory 183  
B. Hidden Line Removal 195  
C. robmod programs 205  
D. CSG Trees and S-Bounds 211  
E. Results for Chapter 8 229  
F. Distance Functions 232  
G. PMC in Four Dimensions 235  

References 241  
Index 246
List of Sections

1. Modelling Solids in Motion 10
   1.1. Purpose of this Thesis 10
   1.2. Contents of this Thesis 11
   1.3. Reader's Guide 13
   1.4. A Short History of Geometric Modelling 14
      1.4.1 Boundary Models 14
      1.4.2 CSG Models 15
      1.4.3 Geometric Coverage of Modelling Systems 17
      1.4.4 Half Spaces 18
      1.4.5 Point Membership Classification 19

2. Adding Motions to Models 20
   2.1. Basic notions 20
   2.2. Regularisation and Triangulability 23
   2.3. Extensions to the Fourth Dimension 24
   2.4. Extrusion 28
   2.5. The Triangulability of Extrusions 30
   2.6. Implementing a Four-Dimensional Model 33
   2.7. Describing Motions 33

3. The Clash Detection Problem 37
   3.1. Interference Detection 37
   3.2. Null Object Detection 40
   3.3. Clash Detection 40
   3.4. Use of Clash Detection 41
   3.5. Related Work 42
   3.6. Multiple Interference Detection 43
   3.7. Four-Dimensional Interference Detection 45
   3.8. Sweeping Methods 46
   3.9. Meta-Level Reasoning 48

4. Basic Concepts and Techniques 51
   4.1. Computational Complexity 51
   4.2. The Bath Group's Algorithms 54
8.7. Implications of this Analysis

9. Multiple Interference Detection
   9.1. Achilles and the Tortoise
   9.2. Divide and Conquer
   9.3. The Basic Minimum Distance Algorithm
   9.4. Improving the Minimum Distance Algorithm
   9.5. Comparisons

10. Four Dimensional Modelling and Sweeping
    10.1. Describing Motions
    10.2. Creating Four-Dimensional Models
    10.3. Four-Dimensional Null Object Detection
    10.4. Examples
    10.5. Extensions
      10.5.1 Improved NOD
      10.5.2 Wider Classes of Motions
      10.5.3 Better Motion Descriptions
    10.6. Sweeping Methods
    10.7. Explicit Sweeping
    10.8. Implicit Sweeping
    10.9. Summary

11. Clash Detection Systems
    11.1. Multiple Interference Detection
    11.2. Four-dimensional Interference Detection
    11.3. Sweeping
    11.4. A Composite Algorithm
    11.5. Complete Clash Detection Systems

12. Conclusions and Further Research
    12.1. Contributions
      12.1.1 Four-Dimensional Modelling
      12.1.2 S-bounds
      12.1.3 Null Object Detection and Interference Detection
Abbreviations and Symbols

\(\mathbf{T}\) tree that represents the null object (bottom)

\(\mathbf{I}\) tree that represents the universal set (top)

\(\mathbf{L}\) location function

\(\text{Ex}\) extrusion operator

\(\text{Sw}\) sweeping operator

\(\circ\) functional composition operator

\(\times\) set product operator

\(\in\) set membership predicate

\(\subseteq, \supseteq\) subset and superset predicates

\(\preceq\) partial ordering on functions: \(f \preceq g \iff f(x) \leq g(x)\) for all \(x\).

\(\leq, \geq\) ordering on numbers

\(\emptyset\) null set

\(\mathbb{N}\) universal set

\(\beta\) bounding function (on CSG trees)

\(\text{Pow}\) powerset function; gives the set of all subsets of a set

\(\mathbb{R}\) set of real numbers

\(\times\) set of interesting values of time

\(\cup, \cap, \setminus\) set operators; union, intersection and difference

\(\text{by}\) set boundary

\(-\) set complement

\(\text{cl}\) set closure

\(\text{in}\) set interior

\(\text{rg}\) set regularisation

\(\mathbb{E}\) probability expectation function

\(\cdot\) scalar product operator

\(\|\|\) Euclidean distance function

\(\text{CSG}\) constructive solid geometry

\(\text{iff}\) if and only if

\(\text{LMC}\) line membership classification

\(\text{NOD}\) null object detection

\(\text{PMC}\) point membership classification

\(\text{w.r.t}\) with respect to
1. Modelling Solids in Motion

This thesis is concerned with the problem of producing and using a computer model that describes a set of solid objects in motion. The problem of storing the shape of a solid object inside a computer has been the subject of much research over the last decade or so, and as a result industrial grade geometric modelling systems are now becoming the respected tools of the draughtsman and the engineer. However research into the area of process planning, such as the automation of robots and machine tools, requires something more than just a description of stationary objects; at the very least it is necessary to know how the objects move in space. Despite this fact relatively little interest has so far been shown in the direct modelling of solids in motion.

1.1 Purpose of this Thesis

A complete solution to the problems of modelling shapes in motion is beyond the scope of any one thesis, and so I have chosen to centre my research around the solution of one particular problem that I call clash detection. Informally, clash detection is the problem of deciding whether a pair of objects with given motions will try to occupy the same space at the same time, and it is obvious that any planning system should try to avoid such a catastrophe. My goal was not just to find a solution to this problem; I also wanted to find solutions that are efficient and to get some idea of which types of solution are best used in which circumstances. On the abstract side this research has lead me to develop a simple mathematical framework that describes shapes in motion, and on the concrete side this has lead me to develop methods for solving clash detection, together with the various computational and conceptual tools that allow these methods to be implemented as computer programs.
1.2 Contents of this Thesis

This research was produced from the starting point of existing geometric modelling technology so that it would be possible to integrate the results obtained into practical systems with the minimum of effort, and a brief introduction to this underlying technology is to be found in section 1.4.

In chapter 2 I am concerned with formulating an abstract, mathematical framework that describes objects in motion; this framework is based on the r-set framework that is commonly used for three-dimensional objects, and I explain how r-sets can be used in four dimensions, where the fourth dimension is temporal. I also discuss how the motions themselves can be described to a computer.

Clash detection is defined formally in chapter 3 and it is highlighted against three related problems, namely interference detection, null object detection, and findpath. Interference detection is the problem of deciding whether two objects in fixed positions occupy the same space at a fixed time; null object detection is the problem of deciding whether a given representation is of the trivial, null object; and findpath is the problem of choosing paths for a pair of objects so that no clash occurs. The three methods that are considered for solving clash detection are then introduced. The first method is called multiple interference detection, in which clash detection is broken down into many calls to an interference detection routine; the second method is called four-dimensional interference detection, in which the clash detection problem is translated into an intersection problem in four dimensions; and the third method is called sweeping, in which clash detection is solved by considering a more complicated interference detection problem in three dimensions.

Chapter 4 introduces some concepts and techniques that have proved invaluable in this research, namely, computational complexity, which allows some quantitative comparisons to be made of the relative efficiency of algorithms; localisation, which allows algorithms to
concentrate on various regions of space; divide-and-conquer, which allows efficient algorithms to be created that can be tuned to improve their performance; and ray-casting, which allows a problem formulated in one space to be rewritten in terms of problems in another space of smaller dimensionality.

Chapter 5 describes the system that has been used as the test bed for this research - a home-grown geometric modelling system called robmod which is based on the shape description method known as constructive solid geometry.

In chapter 6 the notion of S-bounds is described. S-bounds are a new method for creating approximations to objects based on a set of rules for refining approximations, and I see them as one of the most immediately practical results of this research.

Chapter 7 considers the solution to the null object detection problem, and develops a new, efficient algorithm for solving the interference detection problem based on the ideas presented in chapters 4 and 6.

Chapter 8 is concerned with developing some estimates for the computational complexity of part of the null object detection algorithm of chapter 7, and I also suggest a set of possible measures for the complexity of shapes.

Chapter 9 discusses the multiple interference detection method for performing clash detection in some detail, together with an algorithm for calculating the separation between modelled objects, and some experimental results obtained using robmod.

Chapter 10 considers the other two methods for performing clash detection. Four-dimensional interference detection has been implemented in robmod, whereas the sweeping method has only been considered in the light of other published work. These two methods are more difficult to implement than multiple interference detection, and this chapter concludes with some comments on their practicalities.
Chapter 11 is a summary of the results of my study of clash detection itself, together with a list of ways in which the methods studied could be combined into practical systems.

Chapter 12 is a round up of the thesis as a whole, and includes suggestions for future research into the modelling of solids in motion.

1.3 Reader's Guide

The material in this thesis has been ordered so that it should be possible to read it straight through; this section is for the guidance of anyone who wishes to just find the chapters dealing with some specific topics.

The mathematical framework that forms the backbone to this research is reasonably well contained in chapter 2, and anybody who is not interested in the four-dimensional modelling (whose implementation is described in chapter 10) could probably skip this chapter completely. In an attempt to make the main body of the thesis as readable as possible many of the mathematical proofs have been concentrated in the appendices towards the end of the thesis, although chapter 8, on the statistical analysis of algorithms, requires more than a nodding acquaintance with the notions of statistics and probability theory.

Clash detection is treated in chapters 3, 9, 10 and 11, and anybody who is just interested in the basic algorithms should be able to read these chapters and get a rough idea of how the algorithms work; however chapters 4, 6 and 7 (which cover practically all the details of my null object detection algorithm) are probably the minimum prerequisites for anybody interested in actually implementing these algorithms. Similarly, anybody who is just interested in the S-bound approximation schemes should be able to read chapter 6 as a self-contained unit.
1.4 A Short History of Geometric Modelling

I do not intend here to try to provide a full history of the craft of geometric body modelling, nor a detailed description of where it now finds itself; some of this will appear in the sequel, but the interested reader is pointed elsewhere for more detail; for a good, reasonably up-to-date overview I suggest [Requicha & Voelcker 82]; [Requicha 80] covers more detail about representational issues, and [Baer et al 79] covers more on specific modelling systems and, in particular, the way that boundary data is stored.

A solid model of an object has been loosely defined as a representation that permits (at least in principle) any well defined geometric property of the object to be calculated automatically (paraphrased from [Voelcker & Requicha 77]). Whilst there are many systems capable of modelling solids now available, either in the marketplace or in the research laboratory, their abilities and performance are closely linked to the type of internal representation(s) that they use, and the number of radically different representations in use is quite small. In particular, two forms of representation schemes presently predominate, namely boundary representation schemes and constructive solid geometry. Most of the algorithms discussed in this thesis are based on the use of a constructive solid geometry (CSG) representation, although some use is made of a boundary representation for the calculation of the minimum distance between objects. A brief description of these two schemes follows.

1.4.1 Boundary Models

Boundary models have historically been favoured because of their seeming simplicity and the ease with which they may be used to create line-drawing output. A model of the boundary of a solid object must contain information about the faces, edges and vertices in the object, and a common way of arranging this information is to associate a set of linked lists with a model, e.g.
In this example the object node points to a list of the faces which bound it; each face points to a list of edges, which bound it; and each edge has pointers to the (two) vertices which bound it.

Many variations on these schemes exist, depending on the particular implementation (note that the scheme illustrated cannot support faces with 'holes' in them). In particular, extra pointers can be added to facilitate the algorithms that work on the data; a good example of such a scheme is Baumgart's Winged-Edge data-structure ([Baumgart 74]). Boundary models are very useful for computing boundary properties, by which we may include the drawing of objects. Other applications include finite-element mesh generation, mass property calculation and interference detection.

The main problem with boundary models is that they are relatively verbose. This means that they are difficult data-structures to keep up-to-date; also, because they are rarely well-sorted, it can often be necessary to search most of the database for even a small amount of data.

1.4.2 CSG Models

In a Constructive Solid Geometry (CSG) system one has available a set of primitive shapes, such as cuboids and cylinders of variable dimensions, and these primitive shapes may be combined in various ways to form the shapes that we desire. Three combination operators are generally supplied, called union, intersection and difference.
The union operator takes two shapes and returns their set union; that is, the shape that consists of the set of points that are in either of the two shapes. The other two operators are similarly defined, and the effects of the operators on two shapes are shown in figure 1.1; note that union and intersection are commutative operators.

Instead of using the standard set operators, most modelling systems use the regularised set operators; these are formed by applying the appropriate standard operator, and then applying a regularisation operator to the result. The regularisation operator is defined in Chapter 2, and we can describe it informally by saying that it removes any non-solid parts of a set. For example, the regularised set intersection of the two objects shown in figure 1.2 is null (the empty set), and not the two-dimensional plate that is the result of applying standard set intersection.

In this text I use U, A, and / to denote the operators for regularised set union, intersection and difference. Both types of set operations discussed here are examples of Boolean algebras; the subject of Boolean algebras is covered in many texts, e.g. [Halmos 63],

Figure 1.1
and some of the results that I prove hold for any Boolean algebra. (That the set operations are Boolean algebras is shown in [Requicha & Tilove 78].)

A CSG description is normally regarded as a binary tree in which the nodes represent the set operators and the leaves represent the primitive shapes. In practice it is usual to augment the structure with unary nodes that represent the application of a rigid-body transformation to the shape represented by its child, but I will often ignore such transformations for clarity; this simplification is valid as we can always rewrite a tree containing transformation nodes into one that only has rigid-body transformations directly attached to the primitives in the tree, and we can then regard the transformations as extra parameters of the primitives.

The main problem with a CSG representation is that the information about the boundaries of objects is not immediately available; however, this is not always a serious problem.

1.4.3 Geometric Coverage of Modelling Systems

There is a limit to the accuracy with which a modelling system may model an arbitrary shape, and this limit depends mainly on the types
of surfaces that can bound the modelled objects. Many existing modelling systems allow only a fairly limited range of surface types, the most popular being planar, cylindrical, and spherical. Parameterised surface models are geometric models in which a large class of surface types may be described. They are useful in applications where the exact shape of the boundary of an object is important, such as in car body design, but a penalty has to be paid for their use as they generally involve a large computational effort. In the context of this thesis the shape of objects are generally adequately described in terms of the relatively simple surfaces mentioned above, as such surfaces are often the types used for engineered objects, and my own experiments have been conducted with just one type of surface, namely planar surfaces. However most of the algorithms described in this thesis are not dependent on the types of surfaces available, and so it should be possible to implement these algorithms on many modelling systems.

In the final subsections I define a couple of terms that are used throughout this thesis.

1.4.4 Half Spaces

One particular class of shapes that are often used in CSG schemes are the half spaces. These are shapes that are described by the set of points for which a real-valued function returns a non-negative value, e.g.

\[ S = \{ (x,y,z) \mid f(x,y,z) \geq 0 \}. \]

Examples of shapes that can be described in this way are the sphere (e.g. \( -x^2 - y^2 - z^2 + r^2 \geq 0 \)) and the infinite planar half spaces (e.g. \( ax + by + cz + d \geq 0 \)).
1.4.5 Point Membership Classification

A basic and particularly useful function on the models of objects is called point membership classification. This function takes a point in space and a set (the set is usually defined by a model of an object) and returns whether the point is inside the set, on the surface of the set, or outside the set. In a true solid modelling system we should always be able to answer such a question. A general discussion of this and similar questions is to be found in [Tilove 80], and I shall be discussing point membership classification in more detail in various parts of this thesis.
In this chapter I will build up the mathematical framework that will serve to act as an abstract model of space and time occupancy. I do not require this framework to map neatly into a computer implementation; rather, it is there for our guidance and as an upper limit on the quality of any implementation that relies on such a framework. I shall also discuss briefly how one such implementation can represent its models, and finally present some ways by which a human user might wish to describe models to such an implementation.

The mathematical framework that I shall develop is an extension to the triangulable r-set framework used in three-dimensions to define the concept of shape; the triangulable r-set framework was developed in the mid-1970's by Requicha at the University of Rochester [Requicha 77]. The basic idea is that we define a shape as any set of points in space that obeys certain rules, and that these rules ensure that the set has 'solidity'. The definition and the rules are couched in terms of a branch of mathematics called point-set topology. Sections 2.1 and 2.2 briefly describe how the framework is defined in three dimensions, and I then proceed to extend it to deal with objects in motion. Readers who are prepared to take the results on trust may decide to skip these sections, as a thorough understanding of this material is not a prerequisite for reading the rest of this thesis.

2.1 Basic notions

This section is a brief outline of the terms and concepts that will be used in our framework, and is meant simply to reawaken memories in those who have or have had a knowledge of elementary topology. For more details see any elementary text on the subject, such as [Mendelson 75].

†. By "model" I do not necessarily imply a three-dimensional model.
A topological space is an abstract mathematical space in which the important notions include those of open set, connectedness, and compactness. Much has been made of the study of topological spaces simply because so many of the interesting and fundamental properties of common, everyday spaces are mirrored in topological spaces, and so results in topology have relevance in many other fields.

A metric space is a topological space with which is associated a distance function. This function gives you the distance between any two points in the space and also obeys certain rules, such as the triangle inequality ("no side of a triangle is larger than the sum of the other two"). Many results that I derive and quote will be in terms of topological spaces and metric spaces so that any readers familiar with these terms may see the results in full generality. Normally I am only really interested in applying these results to certain Euclidean spaces, and so one may visualise the results in terms of these spaces without any loss.

The spaces that we are interested in are the first few Euclidean spaces, denoted En for some small integer n. Formally En is the set of n-tuples \( (x_1, x_2, \ldots, x_n) \), with each \( x_i \in \mathbb{R} \) (the set of real numbers) and the usual Pythagorean distance function \( d \). In vector notation \( d \) is given by

\[
d(x, y)^2 = (x - y)^2
\]

An infinite line is an example of a set that is topologically equivalent to E1. Similarly an infinite plane is topologically equivalent to E2, and we usually think of our universe as equivalent to E3.

The notion of an open set is usually taken as being fundamental in the definition of a topological space. An example of an open set in E1 is the open interval \((0, 1)\), which is \( \{x \in \mathbb{R} | 0 < x < 1\} \). A closed set is one whose set complement is open, and so the closed interval \([0, 1]\) = \( \{x \in \mathbb{R} | 0 \leq x \leq 1\} \) is closed (in E1) because its complement, \((-\infty, 0) \cup (1, \infty)\) is open. The set union of any number of open sets is always an open set, as is the set intersection of a finite number of...
open sets.

The notion of (arcwise) connectedness refers to whether two subsets of a set are linked in the set - so the points 0 and 4 are not connected in the set $[-\infty,1)\cup(2,\infty]$, whereas the points 3 and 4 are. Compactness is a less intuitive concept in Euclidean space, but it is not an important concept in this discussion and it suffices to note that all closed and bounded subsets of $\mathbb{E}^n$ are compact.

Not all subsets are either open or closed; the null set $\emptyset$ is always open, and the whole of $\mathbb{E}^1$, $(-\infty,\infty)$ is open (in $\mathbb{E}^1$), and as these two sets are complementary then both are also closed. Arbitrary subsets need not be either open or closed; the half-open interval $[0,1) = \{x \in \mathbb{R} \mid 0 \leq x < 1\}$ is an example of a subset of $\mathbb{E}^1$ which is neither open nor closed.

The interior of a set is the largest open set that it contains. I shall denote the operation of finding the interior $\text{in}$, so that $\text{in}([0,1])$ is the set $[0,1)$. The closure of a set is the smallest closed set containing the set. It is obtained with the operator $\text{cl}$, so that $\text{cl}([0,1])$ is $[0,1]$. The boundary of a set $S$ is defined by

$$\text{by}(S) = \text{cl}(S) \setminus \text{in}(S)$$

where $\overline{S}$ denotes the set-complement of $S$. For example, the boundary of $[0,1]$ (in $\mathbb{E}^1$) is the set $[0,1)$, which just contains two points. The boundary of a set $S$ is always well-defined, with $\text{by}(S) = \overline{\text{by}(S)}$, and for $\mathbb{E}^n$ this definition of the boundary corresponds to our natural idea of the boundary. So with reference to the point membership classification function (PMC), which was mentioned in section 1.4.5, we can define the result of applying PMC to a point $x$ and a set $S$ to be INSIDE, ON or OUTSIDE depending on whether $x \in \text{in}(S)$, $x \in \text{by}(S)$, or $x \notin \text{in}(S)$.

The regularisation of a set is defined to be the closure of its interior. As this is a useful concept we define the regularisation operator, $\text{rg}$, by
\[ \text{rg} = \text{cluin}, \]

where \( \Box \) denotes functional composition.

\[ \text{2.2 Regularisation and Triangulability} \]

It can be shown ([Requicha 77]) that regularising a set in \( \mathbb{E}^3 \) removes any non-solid faces, edges or vertices, and so it is a primary candidate as a filter that creates "solid" sets out of arbitrary sets in \( \mathbb{E}^3 \). A set is regular if it is equal to its regularisation, i.e.,

\[ A \text{ is regular iff } A = \text{rg}(A) \]

(\( \text{iff} \) is a shorthand for \( \text{if and only if} \)). If we start with regular sets and always regularise the results of any set operations then we will always be dealing with regular sets, and for this reason regular set operations are normally used in a CSG scheme. It can be shown that both the standard set operations and the regular set operations form examples of Boolean algebras [Requicha & Tilove 78], and I shall exploit the properties that that implies later. Notice that all regular sets are, by definition, closed, and so all bounded regular sets are compact. Bounded regular sets are also called \( r^{-} \)sets.

Although regularity has been emphasised as a good thing for sets representing physical objects to have, one must point out that there are \( r^{-} \)-sets which are not physically realisable because they could not be said to be finitely describable. [Requicha 77] gives an example of such a set; one of the bounding surfaces of the set has the form \( y = \sin(1/x) \). There is another condition that we can place on our (abstract, three-dimensional) framework, and that is the sets it represents be triangulable. Informally, by triangulable we mean that we can divide up surfaces into curved triangles, where each (curved) face, edge and vertex are simply describable. More formally, the

\[ \text{So } f \circ g(x) \text{ is } f(g(x)). \]
subset is triangulable if there exists a planar polyhedron which is composed of tetrahedra, and a homeomorphism (a 1-1 function that preserves open sets) that acts on the polyhedra to give the required subset. All "sensible" regular sets are triangulable, in a sense that will be made clearer in the next section.

2.3 Extensions to the Fourth Dimension

A general transformation of $E^n$ is just a mapping with functionality

$$E^n \rightarrow E^n$$

Of particular interest will be mappings of $E^3$ that preserve distance. Let $d$ be the standard Euclidean distance function in $E^3$. We say that $T$ is a rigid-body transformation if $T$ preserves signed angles and if

$$d(T(x), T(y)) = d(x, y) \quad \text{for all } x, y \in E^3.$$  

Rigid-body transformations are easy to characterise, as they can always be reduced to a canonical form, e.g. a translation in space followed by a rotation about some axis.

To model a moving object I model (the shape of) the object in some rest state, say (but not necessarily) at time $t = 0$, and also create a location function. A location function takes a time and returns a rigid-body transformation, so that its functionality is

$$R \rightarrow (E^n \rightarrow E^n).$$

I will generally denote location functions by $L$. If we have a set $S$ which models an object in its rest state, we may say that the object is modelled at time $t$ by the set

$$L(t)(S)$$

I could have emphasised the fact that $L(t)$ is an entity in its own right (a function from $E^3$ to $E^3$) by writing this new set as $(L(t))(S)$, but I shall follow common mathematical practice and allow
looser notation where it cannot cause confusion.

The location function enables us to replace the standard question that is answered by PMC, viz, "where is the point \( x \) with respect to \( S \)?", by the question, "at time \( t \), where is the point \( x \) with respect to \( S \)?". The answer is

\[
\text{PMC}(x, L(t)(s))
\]

(where PMC has functionality

\[ E^3 \times \text{Pow}(E^3) \rightarrow \{ \text{INSIDE, ON, OUTSIDE} \} \]

and \( \text{Pow}(E^3) \) denotes the set of all subsets of \( E^3 \)).

**Example 1**

Let \( S \) be the set \( \{(x,y,z) \mid x \geq 0\} \) (an infinite planar half-space), and let the object that \( S \) represents have constant velocity \( v \) along the \( x \)-axis, so that \( S \) represents the object when \( t=0 \). Then \( L(t) \) is the translation

\[
L(t)(x,y,z) \mapsto (x+vt,y,z)
\]

and so

\[
(x,y,z) \begin{cases} \text{INSIDE} & \text{the object at time } t \text{ if } x \geq vt \\ \text{ON} & x = vt \\ \text{OUTSIDE} & \text{if } x < vt \end{cases}
\]

But remember that \( L(t) \) is a rigid-body transformation and always has an inverse, and so as

\[
\text{PMC}(T(x), T(S)) = \text{PMC}(x, S)
\]

for arbitrary rigid-body transformations \( T \), points \( x \), and sets \( S \), we have the identity

\[
\text{PMC}(x, L(t)(s)) \text{ is } \text{PMC}((L(t)^{-1}(x), s)
\]

This suggests a way of extending the PMC function to take time into account. For a fixed object at rest we can think of PMC as defining an indicator function.
\( \gamma \colon E^3 \to \{ \text{INSIDE, ON, OUTSIDE} \} \)

But if the object is moving and has a location given by a function \( L \), then by the same argument as above we can define a function

\[
\Gamma : E^3 \times R \to \{ \text{INSIDE, ON, OUTSIDE} \}
\]

by

\[
\Gamma(x,t) = \gamma((L(t))^{-1}(x))
\]

But \( E^3 \times R \) is \( E^4 \), and so if we call the four-tuple \((x, t)\) a point in \( E^4 \) we then have

\[
\Gamma : E^4 \to \{ \text{INSIDE, ON, OUTSIDE} \}
\]

and \( \Gamma \) is a particular extension of PMC to four dimensions which allows time to be taken into account.

**Example 2**

As example 1, with \( S \) the infinite planar half-space \( x \geq 0 \) moving with the constant velocity \((v, 0, 0)\). Then

\[
\Gamma(x, y, z, t) = \begin{cases} 
\text{INSIDE} & \text{if } \{ x-vt \} > 0 \\
\text{ON} & \text{if } x-vt \in \{ 0 \} \\
\text{OUTSIDE} & \text{otherwise}
\end{cases}
\]

The function \( \Gamma \) does describe a moving object in the sense that we can do (extended) PMC with it - but how do we create the \( \Gamma \)? In three dimensions the function \( \gamma \) is related to the regular shape that is the set of points for which \( \gamma \) returns either INSIDE or ON, namely the regular set

\[
\gamma^{-1}(\{ \text{INSIDE, ON} \}).
\]

Similarly we can relate to a given \( \Gamma \) a set in four dimensions that is

\[
\Gamma^{-1}(\{ \text{INSIDE, ON} \}).
\]

\( \dagger \)

\( \gamma^{-1} \) refers to the inverse of \( \gamma \).
What can we say about this set? Firstly, note that we can take a slice through the set at constant time \( t \) say by setting the value of time to \( t \) throughout the equations. Then if we do that we get a set which is \( \mathbb{L}(t)(S) \) — which is regular. Secondly, it is difficult (or, at least, non-intuitive) to draw a picture of these sets in the general case, as that would take a projection of \( E^4 \) onto \( E^2 \). However, in certain cases we can take views that capture the useful information.

Example 3

Let \( S \) be the unit cube, \( 0 \leq x \leq 1, 0 \leq y \leq 1, 0 \leq z \leq 1 \), which lies at that position at time \( 0 \), and let \( S \) have constant velocity \( v \) along the \( x \)-axis. So as the \( y \) and \( z \) coordinates never change we may ignore them, and view the four-dimensional set as in figure 2.1.

![Figure 2.1](image)

This example illustrates that without any limit on the time for which the object exists we produce four-dimensional sets that are unbounded in the \( t \)-direction. Fortunately, we are never worried about the entire past and future of objects, and can agree to begin and end the model at some fixed times.
From now on I will talk about all our time components as coming from some suitable subset of $R$, namely TIMES, which will in practice be a closed interval.

2.4 Extrusion

Definition

The extrusion operator, $Ex$, takes any subset $S$ of $E^3$ and a location function $L$, and produces a four-dimensional set thus:

$$Ex(S, L) = \{(x, t) | t \in \text{TIMES and } x \in L(t)(S)\}$$

So $Ex : \text{Pow}(E^3) \times [R \to [E^3 \to E^3]] \to \text{Pow}(E^4)$

It should be clear that $Ex(S, L)$ is roughly the same as the set $\Gamma^{-1}([\text{INSIDE, ON}])$ that we toyed with earlier, as $(x, t) \in Ex(S, L)$ iff $x \in L(t)(S)$. So what are the characteristics of $Ex(S, L)$?

Theorem 2.1

If TIMES is regular (in $R$), $S$ is regular (in $E^3$), and $L$ continuous, then $Ex(S, L)$ is regular (in $E^4$).

Theorem 2.2

If TIMES and $S$ are closed and bounded, and $L$ continuous, then $Ex(S, L)$ is bounded.

Theorem 2.3

If TIMES is a connected subset of $R$, $S$ a connected subset of $E^3$, and $L$ continuous, then $Ex(S, L)$ is connected (in $E^4$).

The proof of these theorems appears in Appendix A.
So if we are given a subset of $E_3$, which is a regular set model of an object, we can extrude this to a subset of $E_4$ which is a model of the object on the move. Furthermore, under quite acceptable conditions on the motions and TIMES we find that this set is also regular.

Combining the theorems would also tell us that if $S$ is an $r$-set (i.e., regular and bounded), $L$ continuous, and TIMES compact, then $\text{Ex}(S,L)$ is also an $r$-set. The next theorem, which is proved in Appendix A, shows us one way in which we might generate these four-dimensional sets. But firstly, some explanation of the notation is required. By a regularised Boolean function $F$ of sets I mean some function

$$F: T^m + T$$

which is given by some finite combination of regularised set operations over some topological space $T$. In practice, we are interested in $T$ as $E_n$, for some $n$. Given another topological space $V$ then for each such $F$ there is a natural function $G$ which has functionality

$$G: V^m + V$$

which is obtained from $F$ by replacing all the set operations in $F$ (with functionality $T \times T \rightarrow T$) by the corresponding operations with functionality $V \times V \rightarrow V$. It is then tempting to abuse notation and to use the same name for $F$ and $G$, and this I will do in the statement of the next theorem, where $T$ is $E_3$ and $V$ is $E_4$.

**Theorem 2.4 (Distribution Theorem)**

Let $R$ be a set given by a regularised Boolean function $F$ on $E_3$, so that

$$R = F(p_1, p_2, \ldots, p_n)$$

(where $p_1, p_2, \text{etc.}$ are the arguments to the function). Then if $L$ is a continuous location function,
\[ \text{Ex}(R, L) = F(\text{Ex}(P_1, L), \text{Ex}(P_2, L), \ldots, \text{Ex}(P_n, L)) \]

To paraphrase the above, "if an object can be represented by a set combination in a CSG scheme, and the primitive objects can be extruded in this scheme, then the extrusion of the object is the set combination of the extrusions of the primitives."

For primitives this result is trivial, and in figure 2.2 I show the result of extruding some simple two-dimensional shapes into three dimensions. Figure 2.3 shows a (two-dimensional) object with a non-trivial CSG representation and two different ways of generating its extrusion.

2.5 The Triangulability of Extrusions

Recall that a further practical restriction on the class of useful subsets of \( E^3 \) is that they can be finitely described by virtue of their triangulability. As explained in [Hironaka 75], the class of semi-algebraic sets is the class which is the finite closure of the
class of algebraic sets under (ordinary) set union, intersection and complement. The algebraic sets are described by an inequality of the form

\[ f(x) \geq 0, \]

where \( f \) is a polynomial in \( x, y, z \). The class of semi-algebraic sets includes all the regular sets in \( \mathbb{E}^3 \) which are formed using primitives with algebraic surfaces, and so includes practically all the surfaces used in modelling systems.

The class of semi-analytic sets is similar to the class of semi-algebraic sets, the difference being that the primitives can be expressed as half-spaces in real-analytic functions (i.e., expressible as half-spaces which are generated by a polynomial).
as power series in small regions, [Dieudonné 60]). This new class contains more surfaces than we could conceive of using. In [Requicha 77] a theorem is quoted that shows that all semi-analytic (and hence all semi-algebraic) sets are triangulable in E3, and so thus are all sensible r-sets. Thus all sensible r-sets could, in theory, be modelled by some (very large) boundary modeller. So we have the implication

\text{semi-analytic set} \rightarrow \text{triangulable set}

Can we extend these results to deal with our models in E4? The answer is "Yes", but we need to take some care. The theorem that states "semi-analytic implies triangulable" is valid in E4 (again see [Hironaka 75]), and so we might hope that a semi-algebraic set (in E3) would map to a semi-algebraic set (in E4), under reasonable location functions. But consider the set

\[ S = \{ (x,y,z) \mid x > 0 \} \]

which is moving with constant angular velocity \( \omega \) about the z-axis. Then the location function is given by

\[ L(t)(x,y,z) = (x \cos \omega t + y \sin \omega t, -x \sin \omega t + y \cos \omega t, z) \]

and so \( \text{Ex}(S,L) \) is described by the half-space

\[ x \cos \omega t - y \sin \omega t \geq 0 \]

which is real-analytic, not polynomial. It is not clear what conditions on \( L \) are necessary to ensure that a semi-analytic set (in three-dimensions) extrudes to a semi-analytic set (in four-dimensions). According to another result from [Hironaka 75] taking \( L \) to be semi-analytic is not enough to guarantee that such an extrusion will be semi-analytic. However this would be enough to guarantee that such an extrusion is subanalytic; subanalytic sets are a class that contain the semi-analytic sets and are triangulable. Further, as long as \( L \) transforms each individual half-space in the representation of the set to a semi-analytic half-space then the extrusion of the whole set will be semi-analytic, and as \( L \) will generally be a
low-order polynomial in the sines and cosines of some functions of
time I conjecture that semi-analytic sets will generally result.

2.6 Implementing a Four-Dimensional Model

Although I said that the purpose of this discussion was to present
a theoretical (or abstract) model of shapes and motions, it should be
noted that the apparatus that I have developed does point the way
towards implementations of models of these four-dimensional sets.

Firstly, the Distribution Theorem (Theorem 2.4) suggests that we
can construct a CSG description whose primitives are four-dimensional
entities, and that given a CSG form of a shape description we just
make an extrusion of the shape by extruding its primitives. Henceforth, I shall refer to such an extrusion as a worm.

Secondly, if a worm is triangulable then we could, in theory, create a finite description of the boundary of the worm. For those not familiar with topology in higher dimensions; a triangulation in three-dimensions consists of (two-dimensional) faces, which are bounded by (one-dimensional) edges, and these in turn are bounded by (zero-dimensional) vertices; thus a worm would be bounded by (three-dimensional) 'hyperfaces', which are bounded by faces, which are bounded by edges, which are bounded by vertices. In practical terms this means that we have to develop an extra layer of links in the boundary model, which can do nothing to improve its clarity. Also I have given no clues as to how we might create such a boundary model.

2.7 Describing Motions

So far in this chapter I have developed an abstract model of
shapes in motion which is simple and does have nice mathematical pro-
properties. Now I would like to see which types of motion we would like
to model, and in parallel how we might describe these motions to an
implementation of the abstract modeller.
A solid body has six degrees-of-freedom, which can be conveniently broken down into three obvious translational degrees-of-freedom, and three rotational. The latter are often described by means of Euler angles, or as axial components (direction cosines). So a general motion can be expressed, say, as six functions of time. Further, as we can always split motions at arbitrary times we can get away with only considering quite simple motions which are valid over a short time span, and then joining the resultant worms together with a (four-dimensional) set union.

Example 4

Take the unit cube as in example 3, and give it the following motion. At rest position at time 0. Then move with velocity $(v,0,0)$ until time 1. Then with velocity $(-v,0,0)$ until time 2. Extrude the shape over time $[0,2]$ and we get the result shown in figure 2.4 (y and z coordinates suppressed).
What appears to be the simplest method of denoting the location functions would be to write down the motions as six functions of time, probably using expressions composed over an alphabet consisting of the real numbers, operators +, *, -, /, and functions such as sin, cos, tan, sqrt, etc. However, this might be very difficult in practice, as these six functions are often quite complex and not easy to generate.

In many (three-dimensional) modelling systems a common mode of reference to a single location is as a chain of primitive transformations. Standard primitives are translation, with three arguments, and rotation about each axis, with one argument each. An example from the ROBMOD language (which I describe in Ch.5) of a valid transformation expression is the expression:

```
to -1 -2 -3 rotx 45 to 1 2 3
```

This expression means the transformation that takes any point, translates it by (-1,-2,-3), then rotates it by 45 degrees about the x-axis (in the conventional positive sense), and then translates the point by (1,2,3). Another interpretation of this particular expression is as a rotation by 45 degrees about the line (-ω,2,3) to (ω,2,3). The transformation could also be written (approximately) as:

```
rotx 45 to 0 2.707 -0.536
```

but I believe that the earlier expression is more intuitive. Transformation chains do give us an easy way of describing motions — namely, by substituting functions of time instead of constants in the arguments. Experience with simple mechanisms does suggest that this method does provide a useful input form.

Example 5

Consider the half-space x ≥ 0 rotating with constant velocity ω about the z-axis. Then a suitable ROBMOD expression to describe the shape and the motion would be
\[ \text{nhalfx rotz omega*t} \]

where \(\text{nhalfx}\) is the primitive half-space, and \(\omega\) is set to the value of \(\omega\).
3. The Clash Detection Problem

In this chapter I wish to introduce the problem that I have called "Clash Detection". Finding solutions to this problem will occupy much of this thesis, either directly or indirectly, and indeed much of my research into "modelling solids in motion" has been with the goal of solving the clash detection problem in an efficient manner. Clash detection is a problem that arises whenever one wishes to test the set of planned trajectories for some objects to see if they can be physically realised. In this chapter clash detection is defined and related to some similar problems, namely interference detection, null object detection, and findpath. Then I will list the general methods that I have been considering to solve the clash detection problem, and which are considered in more detail in chapters 9, 10 and 11.

It should be noted that there is not yet a consensus regarding the naming of these problems and, in particular, the terms "interference detection", "collision detection" and "clash detection" may be used by other authors to mean different things.

3.1 Interference Detection

The need to perform interference detection comes about when we create a representation — that is, a model — of a pair of objects which we place in some positions in space and ask ourselves whether one of the objects penetrates the other. One situation where we need to detect unwanted interferences between objects is in mechanical design, as we have rarely got the objects to hand to see if the configuration we require is physically possible. Indeed, if a configuration is not possible we may want to redesign some or all of the objects involved. In some cases an experienced designer may be able to assure himself that there will be no interference by simply looking at a set of engineering drawings, but with complicated designs
the designer may start to lose some of the confidence that he has in getting everything right, simply because there are so many lines on the drawings. If the parts have been modelled in a CAD system though it is often possible to juggle the viewpoint around to assure oneself of the lack of interference (figure 3.1). But in many cases the situation is never so clear (e.g., figure 3.2).

The planning of a production process which utilises machine tools or robots is another area in which interference detection may be required, and it has special difficulties because objects may be in awkward positions — see how ordered the situation is in figure 3.1, compared with figure 3.2. Fortunately, it is quite feasible to get a solid modelling system to answer this question for us.

The Interference Detection problem may be formalised as follows:
"Given a set of representations of $n$ objects, $R_1$, $R_2$, ..., $R_n$, which are positioned in space, do any pair of these objects occupy some common space?".

To solve this problem we need to see if any part of any object penetrates another. For this to happen their boundaries must intersect, and so we can call upon a boundary modelling system to solve
the problem. This approach has been tried; examples include [Boyse 79] and [Green 81]. Most of the practical problems tend to be numerical in nature in order to deal with the special cases that can occur, especially when two objects are either almost touching or barely interfering.

With respect to the r-set framework that was discussed in chapter 2 we may note that two sets (representing objects) interfere iff they have some point in common - that is, iff their (ordinary) set-intersection is non-null. If we insist that they must actually overlap before we declare interference then we will require that their regularised set intersection to be non-null. Set intersection is a standard operation in CSG schemes, and so we can see that the interference problem can be reduced to testing whether or not a set is null. This problem has been dubbed Null Object Detection (NOD).
3.2 Null Object Detection

The question that NOD strives to answer is simple: "Does this model represent a null object?". In a boundary modelling system this question is easily answered because an object is null iff its boundary is null ([Tilove 81]). However if this test is being performed in order to test for interference then we generally do not have a boundary representation of the intersection of the two objects' handy, and so we would still have to compute this.

In a CSG-based system we can easily represent the set to be tested, but NOD for this representation is not trivial. The simple-minded approach is to compute the boundary, but we can simplify this by noting that any evidence for non-nullity implies that the set is not null; we do not have to compute the "extent" of a non-null set, rather we just want to know if it contains anything at all. So it is easy to write an algorithm that just halts when it detects any "matter" in the representation it is given. [Tilove 81] is partly devoted to this problem, and also includes methods for speeding up this process. As the ability to perform NOD turns out to be important part of clash detection I have improved on his algorithms; my algorithm for performing NOD will be described in chapter 7.

3.3 Clash Detection

I state the clash detection problem formally as follows:

"Given representations of N objects S_1, S_2, ..., S_N whose locations in space are given by functions \( L_1, L_2, ..., L_N \) of time over a time interval \([t_0, t_1]\), state whether any pair of the objects occupy some common space at any time during this time interval".

Here I have used the location functions first introduced in chapter 2. A location function takes as argument a time, and returns a rigid-body transformation. The idea is that \( L_i(t) \) can be applied in turn to the representation \( S_i \) to give a new representation,
which represents the $i$th object at time $t$. So the functionality of the $L$'s is

$$L: \text{Time} \rightarrow \text{Rigid Body Transformation}$$

I have specified that $t$ is to be taken from some closed time interval - this is not a problem as any closed and bounded set can be written as the union of closed intervals. The reason for specifying a closed interval is purely so that the results of chapter 2 hold, and $[t_0, t_1]$ can be regarded as interchangeable with TIMES there.

3.4 Use of Clash Detection

My original motivation for the study of Clash Detection was its potential in robot motion verification. Robots are programmable devices, and by far the most common way of programming them today is by an operator leading them through a series of motions which the robot can digitise, remember, and repeat - the so called "teach mode" of operation. Teach mode has as important advantages its simplicity and the fact that it is easy to see if a program is correct. It has several disadvantages though, including:

a) it can be dangerous - mistakes can be made which can damage the machine, its surroundings, or the programmer;

b) it can be expensive if the robot is already in full time use, and it has to be withdrawn from service while it is being taught;

b) it can be impossible, because the robot and/or its workstation may not yet exist.

So there is now a great deal of interest being shown in simulating what a robot will do, normally by representing the situations at a graphics terminal. In this mode it is up to the operator to "perform" any required clash detection, but it would be useful to be able to automate this task as the same problems that befall visual interference detection apply, and are compounded by the fact that the
3.5 Related Work

Clash detection itself has not previously been well studied, and I have only discovered a few examples of previous implementations. John Boyse implemented a clash detection routine that could handle a few simple motions [Boyse 79]; Jeanine Meyer has equipped IBM's EMULA robot simulation system with a simple clash detection capability [Meyer 81]; and Mazin Balila has been experimenting with clash detection in some simple robot configurations at the University of Leeds [Balila & de Pennington 83]. None of these systems can claim to be in any way general or capable of generalisation, and one of the reasons for this present study of clash detection was to provide some more general guidelines to anybody who wishes to implement this function.

The findpath problem of A.I. may be stated as follows: "Given a set of fixed obstacles and one movable object, find a path for the movable object from a given initial position to a final position such that this object does not clash with any of the others". Now given the stated motivation for this research it may seem strange that I do not study findpath instead of clash detection, and so I add a few lines of explanation here. The basis of the argument is that I expect most types of robot motion will be obvious enough to be suggested by either a human or a simple computational engine, such as a heuristic-guided planner, and that for these cases finding a solution to clash detection may well be cheaper (computationally) than solving findpath. I will briefly mention two recent examples of attempts to solve findpath to justify this view.

Udupa ([Udupa 77]) worked on planning paths for a robot manipulator and achieved some very useful and impressive results - in particular, we should note his use of a hierarchy of modelling spaces. However, the work was closely tied to a particular manipulator configuration and also does not seem easily extendible to deal with the
sorts of close fits that one finds in assembly work.

Lozano-Perez established the notion of configuration space in order to state a general solution to findpath ([Lozano-Perez & Wesley 79]), but there is great difficulty in applying this type of solution to anything other than simple object classes and types of motion. He and Brooks have recently tried to extend their programs to cover a more general solution set and to use heuristics to speed up the process ([Brooks & Lozano-Perez 82; Brooks 82, 83]), but their practical results to date are still limited to fairly simple situations (an object with "2½" degrees of freedom) and are computationally expensive.

In this work I have been trying to find solutions to the clash detection problem that are computationally efficient. In the future it hoped that the study of clash detection may throw some light onto methods for solving findpath; the sort of information that may be useful here includes the distance between objects, which pairs of objects clash (if any), and the existence of any useful free space.

3.6 Multiple Interference Detection

Now I shall describe some possible schemes for solving clash detection, starting with the one which is probably the most obvious, multiple interference detection. If a human were to sit down at a CAD workstation and satisfy himself that some planned motions were clash-free he would probably do so by checking for interference at some number of points along the paths. So we could get the computer to perform these checks automatically, and to give the result "no clash" if there are no interferences. Figure 3.3 shows an example of a square moving and clashing with a triangle - for clarity I shall often use two-dimensional illustrations in this thesis.

As we are given the positions of the objects as functions of time then we can use time as a parameter with which to index the paths of the objects, and establish the situations at which we will test for interference by quoting the time at which the situation occurs.
The simplest way of choosing the times that we will consider is by uniform spacing between the initial time and the final time, and this is the approach used in the EMULA system ([Meyer 81]). However it is not clear what size of time interval we should use - too large an interval might miss a clash, and too small an interval means a lot of wasted computation. As we shall see in chapter 7 interference detection is no longer a very expensive operation, but neither is it particularly cheap.

In a human-operated system the user could tell the machine what sort of time-spacing to use, and my experiments suggest that if a quick-and-easy drawing operation is also available which shows the situation at some or all of the time-steps then the user might learn to adjust the time-step interactively.

Another possibility would be for the machine to increase the number of time-steps automatically, until the user is satisfied that no clashes are possible. For example, if we are interested in values of time between 0 and 1, then we might test at times 0, 1, 1/2, 1/4, 3/4, 1/8, 3/8, ... This sort of sequence is likely to find clashes faster than a strict linear sequence (although for many applications there are likely to be no clashes to find). But in many respects this type of algorithm is dumb, and I have implemented some more intelligent algorithms which will be discussed in chapter 9.
3.7 Four-Dimensional Interference Detection

Recall the definition of the extrusion operator, Ex, from section 2.4; Ex takes a set and a location function and produces a subset of E4 thus:

\[ \text{Ex}(S, L) = \{(x, t) | t \in \text{TIMES} \text{ and } x \in L(t)(S) \} \]

Now two objects clash (during TIMES) iff there exists some point x such that both occupy x at some time t. But this happens iff \((x, t)\) belongs to the extrusions of both objects - i.e., the objects clash iff their extrusions interfere. I illustrate this in figure 3.4, which shows two two-dimensional objects moving and clashing, as seen by observing their (three-dimensional) extrusions.

![Figure 3.4](image)

So if we can a) model the extrusions of any objects that we are interested in and b) do interference detection between these extrusions then we can do clash-detection. Doing b) is not too difficult - many of the techniques that I have developed for (three-dimensional) NOD are applicable in four-dimensions. The problem, as
we noted in chapter 2, comes in step a). Even fairly simple motions, such as constant angular velocities, give rise to complex hyper-surfaces in a worm (i.e., in a model of the extrusions). Part of the contribution of this thesis lies in the experiments with this method, which are described in chapter 10.

3.8 Sweeping Methods

Given an object which moves along a path in space then the space which the object sweeps out can be thought of as another (static) object. Call this new object the sweep of the first. Then if two objects clash their sweeps must interfere. Figure 3.5 shows a two-

![Figure 3.5](image)

dimensional example. So the basis of a sweeping method is to create the sweeps of the objects, and then to look for interferences.

However there are some problems. The first is that if both objects are moving and their sweeps intersect, then that does not necessarily imply that both objects occupied some piece of space at the same time (see figure 3.6). We can get over this by freezing one object and just moving the other relative to the first (figure 3.7), but this is combinatorially expensive if many objects are moving and
we want to test for clashes between all pairs, as we then have to create as many distinct sweeps of the objects as there are pairs of objects, and creating a sweep can be a relatively expensive operation.
The second and most serious problem is that calculating the sweep of a general object is not trivial, and indeed I know of no algorithms to do this. In both boundary-based and CSG-based modellers all the signs suggest that creating such general sweeps would be very expensive.

One way of at least sidestepping the problem is by making the sweeps implicit rather than explicit, so that, say, a cone which is swept through some complicated path would be represented by an entity whose semantics are "a cone swept through the path ...". Of course we now lose the possibility of using the same interference detection operators that we have for explicit model representations, but this approach can be made to work in some simple cases where explicit sweeping would be very difficult (e.g., [Boyse 79]). Another possibility is to use an approximation to the objects that is more amenable to sweeping; this is the approach used in [Balila & de Pennington 83]. More details will be given in chapter 10.

3.9 Meta-Level Reasoning

So far I have briefly described three different methods that can be used to do clash detection. I have not yet tried to rank these methods in order of some absolute scale of usefulness, simply because each is more or less useful in different situations. Indeed we shall see that all can be useful at different stages within a single clash detection run. What is needed in a practical clash detection processor is for some intelligence to control which of our basic techniques (or their variants) are to be used at which stage - we must employ what is known as meta-level reasoning.

Meta-level reasoning is generally used as a way of directing the search for a solution to a problem by means of some relatively cheap control structure. In the above case, if we want a machine to exhibit this type of reasoning we need to know two things: firstly, when we are likely to gain by switching between the techniques, and secondly, the sorts of heuristics that we can use to estimate the
cost associated with combinations of the techniques.

One cost-function that seems useful is the distance apart of the objects - but more on that in chapter 11. Other examples of metalevel reasoning might study knowledge about motions. For example, the use of a robot dynamics package might allow us to concentrate a search for a clash in the direction of motion. Also note that a human uses words such as "slides" to imply more than just a direction of motion - a set of constraints is also implied.

Another set of parameters that such reasoning can control is the degree of approximation of the models that are used. Models are only representations of objects, which for many real objects are not entirely accurate, although most real objects can be represented as accurately as we like. The problem with representing objects very accurately is that it makes the processes that use these representations slower, and in many cases it makes the process very much slower (I will say more about the speed of algorithms in the next chapter). Conversely, in many cases we can get away with a bad approximation to objects if it gives us a fast, correct answer.

An example that is relevant here is that illustrated in figure 3.8. We want to test for interference between the two objects A and B. Rather than modelling them, we first model a pair of circles, A' and B', that are constructed such that A' contains A and B' contains

![Figure 3.8](image-url)
B. Then we can quickly check that A' and B' do not interfere, and so neither can A and B. Such a bubble test is often used implicitly in geometric modelling systems. In theory we can employ a whole hierarchy of approximate representations for our objects, and also for our motions. In chapter 6 I will describe a novel way of generating suitable approximations (S-bounds).

Finally note that it will often be practical to leave the human operator to provide the meta-level reasoning in a clash detection algorithm; in such a case the human would be "composing" a clash detection algorithm to suit the case in hand. Some examples of such man-machine interfaces will be presented in chapter 11.
4. Basic Concepts and Techniques

This chapter is concerned with reporting some concepts and techniques that have been of fundamental importance in my research; knowledge of the contents of this chapter is assumed in the description of the lower level algorithms that are presented in this thesis. Firstly there is the concept of computational complexity; computational complexity is an interdisciplinary subject that concerns itself with providing some quantitative measure of the "speed" of algorithms (rather than of the speed of programs that implement such algorithms). One of the goals of this research was the creation of algorithms that are computationally efficient, and so the concept of computational complexity has been extremely useful.

The techniques reported here have recently come to the attention of the computational geometry community and cannot be said to be well-established, and for this reason I have decided to give a personalised precis of them rather than just pointing to original references. One set of techniques are based on some algorithms for drawing modelled objects which have been developed by a group at the University of Bath, and the other technique is known as "ray-casting" (or equivalently, "ray-tracing"). All of these techniques are especially suited to working on CSG representations of solids, and I shall explain towards the end of this chapter how these techniques are related.

4.1 Computational Complexity

This section is meant as a brief introduction to the notions of computational complexity; the interested reader may wish to refer to [Aho et al 74] for more details, and the reader with a thorough knowledge of the subject may wish to skip this section altogether.
Consider a simple example of an algorithm; bubble-sort, that sorts an array of \( n \) numbers. I have written the algorithm below in a pidgin-Algol (see [Aho et al 74] for details); this is not meant to be a computer programming language, but it is rather an informal language which one can make up for oneself, and which is meant to express the important features of an algorithm as precisely as necessary, whilst hiding the unimportant features.

```plaintext
procedure Bsort( array, n);
    for i = 1 to n-1 do
        for j = i+1 to n do
            if array[j-1] < array[j] then
                swap( array[j-1], array[j]);
        endfor
    endfor
endproc
```

In this example I have not declared the types of the variables, nor have I elaborated on the action of the routine \texttt{swap}(); both these steps would almost certainly be necessary in a real programming language. Instead I have tried to emphasis the control-structure required. Pidgin-algol descriptions of algorithms will often be used in this thesis.

Now if we ask ourselves the question, "How long will this algorithm take to run?" then we can note several things. Firstly, the time taken will depend on \( n \), which is the number of things to be sorted. Secondly, the control-structure shown will enter the if... clause \( m \) times, where

\[
m = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{1}{2}n(n-1)
\]

and if we make some simple assumptions (such as \texttt{swap}() taking a constant time) then the value of \( m \) gives us some idea of how long the algorithm takes to run. In fact, we would generally say that the algorithm is \( O(n^2) \) as the \( n^2 \) term dominates the polynomial for large
n. This terminology is defined thus:

A function \( g(n) \) is said to be \( O(f(n)) \) if there exists a constant \( c \) such that \( g(n) \leq cf(n) \) for all but some finite (possibly empty) set of non-negative values for \( n \); \( g(n) \) is said to be \( \Omega(f(n)) \) if there exists a constant \( c \) such that \( g(n) \geq cf(n) \) for all but some finite set of non-negative values for \( n \); and \( g(n) \) is said to be \( \Theta(f(n)) \) if it is both \( O(f(n)) \) and \( \Omega(f(n)) \).

More informally, an algorithm is \( O(f(n)) \) if it takes at most \( f(n) \) units of time to run, \( \Omega(f(n)) \) if it takes at least \( f(n) \) units of time to run, and \( \Theta(f(n)) \) if the time that it takes to run is roughly proportional to \( f(n) \). Generally \( n \) is the size of the input to the algorithm, but other parameters or sets of parameters can be specified. This abstraction is useful because it tells us how much longer an algorithm takes to run on larger inputs.

For example, if we have a choice of two algorithms, called algorithm A and algorithm B, then if A takes one minute to run with a particular set of input data, and B takes two minutes to run with the same data, then we might naively reckon that A is a better algorithm. But if algorithm A is \( \Theta(n^3) \) and algorithm B is \( \Theta(n^2) \), and we intend to regularly use a data length that is four times that of the test data, then we might expect algorithm B to take (roughly) half an hour, and algorithm A an hour. This sort of argument particularly applies to algorithms for geometric modelling, where the only thing we are certain about the size of the problems that are presented to such algorithms is that they are growing all the time, as people use them for bigger and bigger projects.

Computational complexity is studied for its own sake as a branch of mathematics, and given a particular problem and a particular model of computation it is sometimes possible to prove a lower bound for the computational complexity of any algorithm that can solve the problem. One important example is the sorting problem, and the lower bound for that problem under the common model of computation (the Random Access Machine) is
Several algorithms exist that exhibit this complexity (e.g., HEAP-SORT).

Strictly speaking all the above examples should be quoted as examples of the time complexity of an algorithm. It is often advantageous to carry out a similar analysis to describe the space used by the algorithm. Space complexity figures can often be calculated - for the bubble-sort example it is \( O(n) \) - but unless stated we take "complexity" to mean "time complexity".

We must also be careful to distinguish between "worst-case" complexity and "expected" complexity. An example may help to make this clearer. If we had some sets of data and we expected most sets to be already sorted and we then wanted to design an algorithm to sort them, then a suitable algorithm would be one that checked to see if the set was already sorted, and applied bubble-sort if it was not. We would expect such an algorithm to do better than the \( O(n^2) \) that it achieves on unsorted sets of data. The expected time-complexity would then be the average time taken over all possible inputs, given the probability distribution of the input data.

4.2 The Bath Group's Algorithms

This section concerns a set of programs which have been designed and implemented at the University of Bath, and which produce fast picture output from a CSG representation of an object ([Woodwark & Quinlan 80]). This work has been extended by me to include other functions, especially the NOD function. Until recently CSG systems have been out of favour because of the time it takes to produce a picture of modelled objects. It was assumed that it was necessary to produce a boundary representation first, and then to display the edges from that.
I shall illustrate here a two-dimensional version of this problem, as this is easier to illustrate on a sheet of paper, easier to explain, and yet contains enough complexity to exhibit the features that hold in higher dimensions. So our CSG scheme in two-dimensions will have linear primitives, and for even greater simplicity I will only have one type of primitive, namely unbounded linear half-spaces (or half-planes), given by equations of the form

\[ ax + by + d \geq 0 \]

It should be clear that in such a scheme we can only create polygons, and an example of such a polygon is shown in figure 4.1 where we create a rectangle by intersecting the four half-planes labelled A, B, C and D (the hatching shows the sides of the half-plane boundaries in which the "matter" lies). Now assume that we wanted to draw the rectangle; that is, to find its bounding edge segments. Then we might proceed as follows.

For each primitive in the tree, split the edge of the primitive up into a set of line segments by using the other primitives.

\[ \dagger \] However the method is not limited to planar primitives.
Then check each line segment generated in this way to see if it is visible. (For this algorithm a line-segment is deemed to be visible if it exists on the surface of an object.)

This algorithm will, for example, split the edge of A into three line segments, and declare only the middle one to be visible. This algorithm is very simple, but it does illustrate a couple of features commonly found in geometric algorithms. Firstly, it is an example of a generate-and-classify algorithm; that is, we generate a superset of the possibly visible lines, and then classify these to see which are visible. Secondly, the sort of classification used is an example of set-membership classification.

Set-membership classification tests have been formalised and studied in [Tilove 80], and are applicable to many geometric problems. The general form of set-membership classification is as follows. "Given two sets, X and S, produce a segmentation of X into the subsets that are inside, on and outside S". I have already mentioned one example of set-membership classification, namely, point-membership classification (PMC), and the specific example that we require for our algorithm is called line-membership classification (LMC). An example of LMC is shown in figure 4.2, where a line is segmented into parts that are inside, on and outside a polygon.

![Figure 4.2](image)

What can we say about the (time) complexity of this algorithm? Well, we are taking each of n half-planes and splitting their boundaries up by at most n-1 lines, and so we will have to classify O(n^2)
line segments. Line membership classification can be done in $O(n)$ time, and so the total complexity is $O(n^3)$. (The analogous result for the three-dimensional case is $O(n^4)$, as then we have to take pairs of the faces, which bound half-spaces, to form candidate edges). This is obviously a lot slower than the algorithm given a boundary model - a boundary model of a two-dimensional object is basically just a list of its edges, and so we can draw the object in $O(n)$ time, where $n$ is the number of edges in the representation.

What the Bath group did was to consider a slightly modified problem - namely, the drawing of the object as viewed through a finite window on the infinite plane. This is reasonable, because our graphics devices always do just show a window over the world. So say that we wanted to draw the object from figure 4.1 as seen through the window in figure 4.3a; then we can see that the result is the same as the "sum" of the views through the four sub-windows, NW, NE, SW and SE (shown dotted).

If instead of solving the large problem we solve the four sub-problems (that is, the views through each of the windows) in such a way that we use the two half-plane representations that are valid in each window (figure 4.3b shows a representation that is valid in the window NW) then we end up with four problems that are significantly
simpler than the original one. This is the idea behind the \textit{divide-and-conquer} paradigm; the problem in this case is to produce the simplified models for use in the sub-problems. The Bath group solved this by the observation which I will detail in the next few paragraphs.

Consider the original four-primitive tree (as in figure 4.1) and the window NW. The boundaries of the primitives C and D do not intersect the window, which implies in both cases that the half-planes give a constant contribution to the shape as seen through the window (NW). In fact, they can both be replaced by an infinite object, which I represent by $T$, and which contributes matter everywhere. This can be achieved as a tree substitution, as in figure 4.4a, in just one tree scan (i.e., $O(n)$ time). Now we can take the tree of figure 4.4a, and argue as follows. The subtree $(TAT)$ has leading operator $A$, and has as one of its branches a $T$ (in fact both branches are $T$, but this is irrelevant). This implies that the $T$ branch is redundant, and that this subtree can be replaced by the other branch, to give the smaller tree in figure 4.4b. Applying the same rule a second time gives the tree in figure 4.4c.

If instead of a particular primitive covering a window it is not seen at all then we can replace it in a tree by $\bot$, which represents the null object. The complete set of \textit{rewrite rules} for trees
Figure 4.5

containing \( \top \) or \( \bot \) are shown in figure 4.5 - here \( X \) stands for any subtree, and the operation \( \sim \) stands for the inverse tree, that is, the tree that represents the set-complement of the original tree. This can also be generated by rewrite rules (commonly called De Morgan's Laws), as shown in figure 4.6, where \( X \) and \( Y \) stand for arbitrary trees.†

There is one final trick that I must mention before I have finally described the basic algorithm. I have already described how we can split the problem into four (hopefully) simpler problems. In practice we could split the problem into as many sub-problems as we want (the original Bath algorithm splits a three-dimensional "window" into eight octants). In fact there is no reason why we should just split the original window once, as the full, recursive procedure below illustrates.

†. The reader may note that the operator set \( U, A, / \) and \( \sim \) is not independent. By rewriting CSG trees with a smaller set of operators smaller programs to implement these algorithms result.
procedure Bath_draw( window, tree)
    ntree = simplify( window, tree)
        /* simplify tree w.r.t. window */
    if cando( ntree) then draw( window, tree)
        /* cando() is discussed below; it returns a truth value.
        draw() is the simple-minded routine described earlier — it is given a window in which to clip its output */
    else
        begin
            split window into W1, W2, W3, W4;
            foreach Wi do Bath_draw( Wi, ntree);
        end
end

Here the number of times that the procedure recursively subdivides the problem is controlled by the predicate cando(). cando() can, for example, be arranged to count the number of primitives in the current tree and to only allow draw() to be called when the number of primitives is less than some preset value. This would effectively split the original problem up into a number of constant time problems, but then the cost of the splitting algorithm (Bath_draw()) tends to dominate the total cost of the algorithm.†

In practice cando() is made a function of the size of the window as well, so that when the depth of recursion is large it can allow larger trees to be drawn immediately (with draw()), and this seems to give a good performance. It is also worth noting that this type of algorithm tends to concentrate its attention on the space near the complicated parts of the objects — i.e, those parts where many primitives are close together.

†. But keeping the number of primitives fed to draw() bounded does have the practical result of allowing a program implementing this routine to use fixed-length storage.
4.3 Ray-Casting

The technique of ray-casting has only recently been publicised (e.g. [Roth 82]), but it has since attracted great interest as a means of visualising CSG-defined objects, and indeed in some quarters now seems to dominate as a means of creating graphical output. The reason for this sudden popularity is the extreme simplicity of the basic technique, coupled with the rapidly decreasing cost of computing power.

Ray-casting may be illustrated as follows. Consider figure 4.7, in which we have a sideways view of an observer looking at a scene in

![Figure 4.7](image)

which there are some objects. The objects are illuminated and are typically multi-coloured. If the number of rays that are entering the observers eye is limited we could trace back along each possible ray (using the reversibility of light) to see which surface we find first. Reflecting back from this surface (using some suitable model of surface reflectance) to a light source (or sources) will tell us whether this surface is in fact illuminated, and so whether or not a ray of light does in fact enter the observer's eye, and if so, what colour and/or luminousity it is.

In computer graphics our output devices are invariably of finite resolution, and so the necessary assumption (of a finite number of possible rays) does hold. Ray-casters can be made to simulate very
complex reflectance characteristics, such as shadows, translucency, and highlighting, but I shall concentrate on the cheaper versions, which use a very simple reflectance model to still produce adequate results.

Now consider the actual mechanism for tracing back a particular ray in a CSG context. Again I shall revert to a two-dimensional example (figure 4.8), which could also be regarded as a slice through a three-dimensional example. In it our hypothetical observer is looking along a ray at our rectangle from figure 4.1, which has been twisted to make the example interesting. The ray intersects with the half-spaces A, B, and C at α, β, and γ respectively. Only β and γ are intersections with the rectangle, and we would like the ray-caster to return the first intersection together with the half-space found there. This can be achieved by a slight modification of the general line-membership classification test ([Tilove 80]), but it is worth noting that for our case an easy algorithm exists. The problem with general LMC is that we have to store what is known as neighbourhood information to ensure that correct results are given for the "on" part of the result. In our case we are only really worried
about the "in" part of the result - i.e., we want to know when a ray enters a particular piece of matter. In our example this happens between $\beta$ and $\gamma$. But it is easy to check that $\beta$-$\gamma$ is an inside segment, because as a segment it does not meet any other boundaries of half-spaces and so we only have to test the segment at a convenient interior point - say its mid-point, $\psi$. This is point-membership classification (PMC), and as we are only interested in whether the result is "in" we can simplify the test. This algorithm for casting a ray is shown below.

```plaintext
procedure Ray_cast(ray, tree)
    intersect_list = [{(-\infty, 1), (\infty, 1)}];
    /* the intersect list is a set of pairs of intersection points and which face caused them. */
    foreach primitive P of tree do
        begin
            /* find the intersection points of ray with P -
                for general primitives there may be more than one. */
            foreach intersection $\delta$ of ray with P do
                add the pair ($\delta$, P) to intersect_list;
        end

    sort intersect_list w.r.t points of contact
    /* now the list is an ordered list of contact points */

    last = headof(intersect_list)
    /* headof() extracts the first element of a list,
        leaving the tail. */
    while intersect_list = {} do
        begin
            new = headof(intersect_list);
            set $\psi$ to the mid-point of the intersection-point components of last and new;
            if inside($\psi$, tree) then
                return last;
            last = new;
        end
    /* if we reach here, then there was no intersection */
```
return last;
endproc

Boolean procedure inside( point, tree);
  if tree is a single primitive then
    return p_inside( point, tree);
    /* p_inside() solves the problem for a primitive
     * it is highly dependent on the class of primitives,
     * but quite straightforward. */
  else
    begin
      op * head node of tree;
      ltree * left subtree of tree;
      rtree * right subtree of tree;
      lresult + inside( point, ltree);
      rresult + inside( point, rtree);
      switch op of
        case U: return lresult or rresult;
        case A: return lresult and rresult;
        case /: return lresult and (not rresult);
      endsw
    end
  endproc

inside() is an example of a structural divide-and-conquer routine which uses merge (or combine) rules to combine results from different levels in a CSG tree. Ray_cast() is not complete - for example, I have not specified what to do when more than one primitive intersects ray at a particular point. Also, note that inside() will only work when the point to be tested is either in or outside all of the primitives - an example of a case where it might fail is shown in figure 4.9, where X marks the tested point. Then inside would return true, whereas the point is really on the object A/B. This is not a problem in Ray_cast() because the tested points are never on primitives, but we might in practice get numerical problems - again, when two primitives meet or almost meet at a ray. Solutions to these special cases
are highly implementation dependent, and I shall not discuss them further.

What can we say about the complexity of Ray\_cast()? We can split the algorithm up into 3 steps, namely a) the calculation of the points of intersection, b) the sorting of these points and c) the classification of them. Step (a) is $O(n)$, and step (b) $O(n \log n)$ (with a suitable sorting algorithm). Step (c) consists of potentially $O(n)$ classifications of a point in an $n$-primitive tree, so this step is $O(n^2)$, and predominates. But this is the worst-case complexity - in practice we can find the first surface intersected in fewer than $n$ classifications, and then stop. So the expected time complexity of Ray\_cast may sometimes be better than $O(n^2)$. A further amendment can also be made to Ray\_cast to help speed things up, and I shall discuss this when I come to describe the use of S-bounds (section 6.4).

4.4 Summary

In this chapter I have described several concepts that have been extremely useful in this research. One concept was that of computational complexity; computational complexity is a useful tool for quantifying what is meant by a fast algorithm. Another concept was that of set-membership classification, and this forms the basis of many of the operations that are required as parts of geometric
algorithms. In particular two forms of set-membership classification have been emphasised, namely line-membership classification (LMC) and point-membership classification (PMC), and it was pointed out that for many cases of practical importance relatively simple algorithms for performing these functions can be created.

With the description of the work of the Bath group came the idea of using a simple procedure (simplify()) that will take a CSG tree that defines a shape and form a smaller CSG tree that also describes the shape within a small region of space (a "window"). At the University of Rochester Tilove has coined the word localisation to refer to this concept, and I have reproduced his definition below:

"A localisation of a regular set S w.r.t a regular set C is a regular set, S', such that SAC = S'AC".

(Definition 2.2, p.23, Tilove 81)

Here C corresponds to the "window" which is given to simplify() as one of its arguments, S corresponds to the CSG tree that is the other argument to simplify(), and S' corresponds to the result of simplify().

The other important point about the Bath group's algorithms is the use of the divide-and-conquer paradigm and the predicate cando() to decide dynamically (i.e., while the algorithm is running) how to apportion the effort of the algorithm between "dividing" and "conquering". This enables programs that implement this algorithm to be tuned to reduce the running time of these routines.

Ray-casting is a significant technique because it reduces the problem of detecting the surfaces of a three-dimensional modelled object to that of detecting "surfaces" in many one-dimensional entities, i.e., the rays. This idea of dimensional reduction allows a ray-casting routine to use line-membership classification (LMC), and LMC can itself be cast in terms of point-membership classification.
It is interesting to note that the algorithm of the Bath group and the standard ray-casting algorithm have, conceptually, much in common. I gave only a simplistic view of ray-casting in which a single ray is generated for each pixel (picture element) on the graphics device used. As this can typically be something like $1000 \times 1000$ points this implies a lot of computation. Much of the computation seems to be wasted, because in most cases two adjacent rays will return the same surface. This is a phenomenon called area coherence, which means that most of a picture tends to look locally the same. One simple solution is to just only sample the possible rays (say by taking every $10 \times 10$ points), and then to do a full test locally only if two adjacent sample rays return different surfaces. This gives good results, but there is the possibility of missing small slivers of detail that are not hit by any of the original sample rays. So a more intelligent solution is to get a ray to "hit" a surface, and then spread out to see if there are any slivers nearby.

The situation is rather akin to throwing bags of flour at a (hypothetical) invisible target - not only do you see the point where the flour bag lands, you also see the area where the flour scatters (for more details, see [Roth 82]).

![Figure 4.10](image-url)
Now consider looking at a three-dimensional scene through a grid, as in figure 4.10. The total picture is the sum of the pictures as seen through the grid holes, and so we can hypothesise a Bath-like algorithm that solved this problem - by dividing in the plane of the grid to get smaller problems in the projections of the grid-holes (The Bath group have a better algorithm to do this, as described in [Woodwark & Quinlan 82]). This algorithm would take a large problem and split it up, normal to the rays coming from the scene, until it had sub-problems that it decided (dynamically) were worth solving immediately.

This is slightly akin to throwing a large sheet over an invisible object to make the outline of the object visible. The intelligent ray-caster, however, takes lots of smaller problems (LMC's) which it expands (the flour-bags on the invisible surface) to get a smaller number of larger problems. Thus I claim that both are manifestations of a similar mechanism - in one case, we throw the sheet and "pat it down" over the complex parts of the objects, and in the other we throw the flour bags which explode to find the fine detail in the object.

I have found that such a conceptual unification of these techniques is of great help when one wants to design new, efficient algorithms that work on CSG data-structures.

4.5 Hidden Line Removal

The concepts that have been described in this chapter have been used in many of the algorithms in the geometric modelling system rob-mod; one example of such an algorithm is an algorithm to draw pictures of modelled objects with their hidden lines removed, and this algorithm is described in Appendix B.
5. robmod

The previous chapter was concerned with some of the basic concepts that I have used in this work; in this chapter I will talk about the particular tools that I have developed. I do not want to bore the reader with long descriptions of the hardware used because that is largely irrelevant, but I will want to go into the software developed in some detail. The heart of the system is a program, called robmod, which started off as a geometric modelling program but has evolved into something a little more than that; robmod runs a language, which I shall call ROBMOD, which started off as a way to store shape descriptions, but has grown into an algorithm development tool.

5.1 The birth of robmod

When I started to look into the field of geometric modelling I decided to actually try experimenting with the creation of a "toy" modelling system in order to see what the issues really were. That was BRIDGET ([Cameron 81]), and it brought to me forcibly the problems of creating and maintaining a boundary-based modeller. At the same time the work of the group at Bath came to my attention, and I was quickly able to use their ideas to build a small (if slow) very basic modelling system. This system was rewritten and improved as the first robmod system, which basically did little more than to draw all of the edges in a CSG description of a shape (i.e., the wireframe). A basic block diagram of the system was thus

```
parser/ command interpreter \rightarrow shapes \rightarrow drawing routine
```

The parser/command interpreter would parse an input stream (which contained shape expression statements) and build up explicit CSG tree
structures. When a \texttt{draw shape_expression} command was issued the expression tree was created, and a new tree created from that which was then fed to the shape drawing routine. A very simple example is

\begin{verbatim}
a = cube 1 2 3
b = cube 2 3 1
c = cube 3 1 2
draw a+b+c
\end{verbatim}

shown in figure 5.1, together with its output. The assignment statement binds \texttt{a} to a shape expression, which in this case is a primitive parallelepiped. A shape expression is a CSG expression, augmented with transformation expressions and with other variables as possible primitives. A transformation expression is a chain of primitive transformations - examples of primitive transformations are \texttt{to 1 2 3}, which means a translation by \((1,2,3)\), and \texttt{rotx 15}, which means a rotation about the x-axis by 15 (degrees). Examples of primitive shapes are the parallelepiped, the cylinder (e.g., \texttt{cyl 2 5}) and the cone (e.g., \texttt{cone 1 4}). The set operations are denoted by \(+, \ast,\) and \(\div\), for regularised union, intersection and difference respectively. The first mechanism for creating the CSG tree was similar to a macro substitution mechanism - the example in figure 5.1 might just as well have been drawn by the command \texttt{draw cube 1 2 3 + cube 2 3 1 + cube 3 1 2}. Other legitimate sets of ROBMOD commands include

\begin{verbatim}
Example 1
a = cube 1 2 3
draw a/(a rotx 60)

Example 2
arm = cube 5 1 1
hand = cube 2 1 1
robot = arm + (hand rotx 30 to 4 1 0)
\end{verbatim}
In the first example, macro substitution gives

\[(\text{cube } 1 \ 2 \ 3) \div (\text{cube } 1 \ 2 \ 3 \ \text{rotx } 60)\]

which is converted into the explicit tree

```
/  
\|  
\|  
---\text{cube } 1 \ 2 \ 3\ |  \text{rotx } 60\ |  \text{cube } 1 \ 2 \ 3
```

Then the system creates another explicit tree which is in the form accepted by the drawing routines. These routines take a tree with only union and intersection operators, and with the leaves as only planar half-spaces - i.e., of the form

\[ax + by + cz + d \geq 0\]

This is done by substituting a six half-space tree for each instance of `cube`, and then using the rewrite rules

\[
\begin{align*}
X/Y & \rightarrow X\overline{Y} & \overline{X/Y} & \rightarrow \overline{X}U
\end{align*}
\]

\[
\begin{align*}
\overline{XUY} & \rightarrow \overline{X}U
\end{align*}
\]

The cylindrical and conical primitives are approximated by polyprisms at this point and the resultant tree for \[(\text{cube } 1 \ 2 \ 3)/(\text{cube } 1 \ 2 \ 3 \ \text{rotx } 60)\] has the structure shown in figure 5.2. robmod is written in the C programming language ([Kernighan & Ritchie 78]), which includes features such as dynamic storage allocation, recursion, and explicit storage pointers that made this use of explicit tree structures an easy and natural way to program. The original system was written on a DEC PDP 11/60, but has since been moved onto a VAX 11/750 and an ICL Perq; both running under the UNIX® operating system. The parser
was written with the help of the YACC compiler-compiler ([Johnson 78]), which greatly simplifies the creation of expression trees.

5.2 Further Development

The parts of robmod that I have described so far make up a system that is rather akin to a geometric "pocket calculator", in that you feed the system with a shape expression, and it draws you a picture. It was felt desirable to be able to get the system to save, recall, and reuse shape descriptions and actions, and this was achieved in a way akin to the BASIC programming language. In BASIC a command can either be interpreted immediately (e.g, print "hello") or it can be added to a program simply by proceeding it with a number (e.g, 10 print "hello"). I have used exactly this format to give a very simple program structure - programs can be edited simply by reentering a line, and they can also be saved in a file for subsequent reuse.

†. UNIX is a Trademark of Bell laboratories.
The next stage in the development of robmod came with the addition of new functions that acted on shape expressions - noticeably, an explicit NOD routine and an associated interference detection routine. At this point it was decided to generalise the assignment mechanism to give variables a type attribute which could be used to assist the ROBMOD parser and also to provide some degree of interactive syntax checking. The type attributes now available to robmod variables include

real, trans(f)ormation, shape, Boolean, body, assembly, worm, motion

so that, for example, the expression

\texttt{cube a 2 3}

is valid when \(a\) is assigned to a variable of type \texttt{real}. This is particularly useful in expressions such as

\texttt{cyl 2 5 to 0 0 (t*t)/2}

as this expression can be said to describe a shape and a motion. (The type \texttt{motion} gives an alternative way of defining a more restricted class of motions, and is described in chapter 10.)

This mechanism is implemented by adopting a general assignment operator that can be extended to cover new types easily, and an expression evaluation module that contains a set of typed functions. For example, the arithmetic addition operator, \texttt{'+'}, is described by a function with type

\texttt{+: real x real + real}

Under this scheme only numbers are primitives that can be entered by the user - even the shape expression \texttt{cube 1 2 3} is created by a call to the function

\texttt{cube: real x real x real + shape}
An example of a more exotic function is

\[ \text{interfere: shape \times shape \rightarrow Boolean} \]

where \( \text{Boolean} = \{ \text{true}, \text{false} \} \). The type \( \text{Boolean} \) is used in the conditional jump statement, which has a simple syntactic form (if \( \text{Boolean expression goto linenumber} \)). This gives the ROBMOD language the ability to evaluate simple functions, with some functions, such as \text{draw}, having a "side-effect" of producing graphical output.

The other types shown also need some explanation. \text{worm} and \text{motion} are used only for the four-dimensional experiments that I have been doing, and will be explained in chapter 10. With regard to \text{body} and \text{assembly}, it was realised that it would be useful to have a class of "finished" shapes - that is, shapes that are no longer to be changed - and to associate with them extra information - including a wireframe. This is the \text{body} type - the only body expressions allowed are one to create a (fixed) body from a shape, and one to read a pre-calculated body description from a file. Assemblies are ways of collecting aggregates of bodies together, but are not fixed in the way that bodies are, for they can be modified by transformation expressions that contain \text{real} variables. An example of an assembly in use is in the following actual ROBMOD program, whose associated output is shown in figure 5.3. Changing the values of "x\_coord" or "y\_coord" will change the assembly referenced by "ass" without the need to reassign it.

```
10 rem this is a comment line
20 x\_coord = 5
30 y\_coord = 10
40 table = cbody( cube 20 20 1 to 0 0 -1 )
50 rem cbody creates a body
60 puck = cbody( cyl 2 1 )
70 rem this creates the puck which slides on the table
80 ass = table @ ( puck to x\_coord y\_coord 0 )
90 rem this creates an assembly
100 rem varying x\_coord and y\_coord will 'move' the puck
```
110 wire ass
120  rem this draws the wireframe of the assembly, which is just
130  rem the wireframes of the composite bodies, suitably moved.

Figure 5.3

Another program fragment will illustrate the use of the conditional jump statement - it is a simple form of the multiple interference detection algorithm described in Ch.3, and its associated output is shown in figure 5.4.

10 $t = 0$
20  rem initial time
30 height = $- \left( \frac{9.806 \cdot t \cdot t}{2} \right) + 3$
40  rem height gives the height of the block above the ground
50 Block = cube 1 1 1 to 0 0 height
60 ground = phalfz
70  rem phalfz is the infinite primitive, $z \leq 0$
100 if interfere ground, Block goto 200
110  rem if the block hits the ground, jump out of the loop
120 $t = \text{eval} \ t + 0.1$
130  rem eval forces immediate evaluation of a constant expression
140 goto 100
Note that the expression evaluations are delayed until they are used to give some form of user-definable functions. In computer-science terminology, the right-hand sides of assignments are treated rather like lambda expressions, instead of constants—also known as a "call-by-name" system. Also note the use of the special function `eval` to force immediate expression evaluation in line 120—without it the assignment "t = t + 0.1" would not make sense.

This example also shows how the ROBMOD language has been used as a target language for some of the algorithms described in this thesis. The general philosophy has been to create a language with enough power to test my algorithms; once the algorithms have been usefully developed, they should then be provided in a more efficient form as primitive functions in the language itself.

This concludes my overview of ROBMOD—more examples of ROBMOD programs are to be found in Appendix C.
6. S-Bounds

In chapter 3 the use of bubble testing and box testing in geometric algorithms was noted; such tests are used as parts of algorithms because it is easy to detect whether two bubbles or boxes intersect, and only if they do intersect need one examine their contents closely. In CSG based algorithms the boxes and bubbles are often placed around the primitive objects in the CSG representation. This chapter is concerned with a new method that I have developed for generating such bubbles or boxes around each node in the CSG tree; I have called this method the S-bounds method. S-bounds take into account the structural information that is present in a CSG description to produce tighter bounds for an object than are normally generated in a standard bubble or box test. S-bounds are especially suited to problems that involve the testing of entities for intersection, and are used to great effect in the NOD algorithm which is described in chapter 7.

6.1 Constructive Solid Geometry Trees

A CSG representation is normally something equivalent to a binary tree, in which the nodes represent set operations and the leaves represent primitives. So if A, B, C are primitives then we can
represent the Boolean expression \( AA(B \cup C) \) by the tree in figure 6.1. I shall want to assign labels to trees, and also to subtrees, and the way that I shall do this is shown in figure 6.2. Then I shall say that \( T \) is the tree \( AA(B \cup C) \), \( T' \) is the tree \( B \cup C \), and that \( T' \) is a subtree of \( T \).† As has probably become apparent I shall want to talk about trees a lot, and also about substitutions on trees. A substitution on a tree is a change to a tree to produce a new tree. For example, suppose we wanted to change the subtree \( T' \) to the set union \( T' \cup D \), where \( D \) is a terminal symbol. Then we get the tree in figure 6.3 which we can represent by the expression \( T[T' \leftarrow T' \cup D] \) to mean "the tree formed from \( T \) by replacing \( T' \) by \( T' \cup D \)". (This notation follows that used in [Tilove 81].) I do not want to labour this point too much, but it should be noted that the substitution

\[
T[T' \leftarrow T' \cup D]
\]

do not make (syntactic) sense, in the same way that a programming language assignment statement does. However, we must take care that our trees do remain trees, and do not become directed graphs, which would leave us open to definitions like

†. Although in the context of this research I am using \( U \), \( A \) and \( / \) to denote the regularised set operations the results of this chapter are only dependent on the operators having their usual meanings in a Boolean algebra.
Figure 6.3

\[ T = T' \cup A \quad \text{where } T' = TAB. \]

With this proviso in mind I shall extend the notation thus:

\[ T[T_1 \cup T'_1; T_2 \cup T'_2] \] is to be \( T'[T_2 \cup T'_2], \)

where \( T' \) is the tree \( T[T_1 \cup T'_1] \), etc.

6.2 Semantics of Trees

So far I have only defined a class of symbol structures - now to give them some meaning. If \( T \) is a tree then its terminal symbols either represent primitives or other trees. An interpretation of primitives is a mapping

\[ i: \text{Primitives} \rightarrow \text{Sets} \]

so that \( i \) associates with each primitive object a set, which is normally a regular set. Given such an interpretation of the primitives we can extend it to an interpretation of trees (whose terminal nodes represent primitives) by defining a function

\[ I: \text{Trees} \rightarrow \text{Sets} \]
by the usual rules - viz.

(1) If T is a primitive then
    \( I(T) = i(T) \).

(2) If T is of the form AUB then
    \( I(T) = I(A) I(B) \).

(3) If T is of the form A\&B then
    \( I(T) = I(A) \land I(B) \).

(4) If T is of the form A/B then
    \( I(T) = I(A) / I(B) \).

Practical representations of CSG trees may often contain nodes denoting other entities, such as nodes that denote transformations, and nodes that denote that a copy of another named tree is to be inserted. It is always possible to rewrite such trees without these nodes and I shall assume that this has been done.

In what follows I shall denote individual interpretations (of primitives) by superscripts and use the same superscripts to distinguish the extensions of these interpretations to trees. For example, \( I' \) will be the extension of the interpretation \( i' \).

6.3 Bounded Trees and S-bounds

I define a bounded tree to be a triple, consisting of a CSG tree (\( T \) say), an interpretation of primitives (\( i \) say), and a bounding function

\[ \beta : \text{subtrees of } T \rightarrow \text{sets} \]

which associates with every subtree of \( T \) (including \( T \) itself) a set, which I call a bound. In practice I will often talk about the bounded tree \( T \), in which case the interpretation and the bounding function will be taken for granted.
I also define the bounded interpretation of a bounded tree to be a function

\[ I_B : \text{Bounded Trees} \to \text{Sets} \]

which is given by the following definition:

1. If \( T \) is a primitive then
   \[ I_B(T) = i(T) \land \beta(T). \]

2. If \( T \) is a tree of the form \( AUC \) then
   \[ I_B(T) = (I_B(A) \cup I_B(C)) \land \beta(T). \]

3. If \( T \) is a tree of the form \( ALC \) then
   \[ I_B(T) = (I_B(A) \land I_B(C)) \land \beta(T). \]

4. If \( T \) is a tree of the form \( A/C \) then
   \[ I_B(T) = (I_B(A) / I_B(C)) \land \beta(T). \]

These rules are just like the rules for the standard interpretation, except that the set that results at each node is further intersected by the bound at that node, as given by the bounding function. The reason for this definition is that if \( T' \) is a subtree of \( T \) then \( \beta(T') \) is going to be chosen to be a set that can be used as an approximation to \( I(T') \), in the sense that we only need to "worry" about \( I(T') \) within \( \beta(T') \). The bubbles or boxes that are used in many geometric modellers are bounds around the primitives in this sense.

I will introduce here a notation that I use in the following sections. If \( \beta \) and \( \beta' \) are bounding functions on a tree that are identically equal except on one particular subtree \( T' \), i.e.

\[ \beta'(T) = \begin{cases} S & \text{if } T = T', \text{ for some set } S \\ \beta(T) & \text{otherwise} \end{cases} \]

then I say that

\[ \beta' = \beta[T' \# S]. \]
Here "#" is just used as a separator. I shall also extend this notation by defining

\[ \beta[T_1\#A;T_2\#B] = \beta'[T_2\#B], \quad \text{where } \beta' = \beta[T_1\#A]. \]

**Definition 6.1**

Define a partial ordering on functions that return sets by

\[ f \sqsubseteq g \quad \text{if} \quad f(x) \subseteq g(x) \quad \text{for all } x. \]

(Equivalently, we say that \( g \sqsupseteq f \). Note that I am using "\( \subseteq \)" for the subset predicate.)

**Definition 6.2**

Let \( \psi \) be a tree with bounding function \( \beta \). I say that \( \beta \) is a consistent bounding function if

\[ I_{\beta}(\psi) = I(\psi), \]

and I say that \( \beta \) is a totally consistent bounding function if for all \( \beta' \sqsupseteq \beta \)

\[ I_{\beta'}(\psi) = I(\psi). \]

My target is to generate totally consistent bounding functions for CSG trees; the next definition introduces a subclass of the class of totally consistent bounding functions that allows this to be done in a straightforward manner.

**Definition 6.3**

Let \( \psi \) be a bounded tree with bounding function \( \beta \). Then I say that \( \beta \) is an \( S \)-bound function on \( \psi \) if for any other function
\textbf{Theorem 6.4}

Let \( \Psi \) be a bounded tree with a bounding function \( \beta \) that satisfies

\[
\begin{align*}
\beta(T) &> i(T) & \text{if } T \text{ is a primitive of } \Psi, \text{ and} \\
\beta(T) & = \emptyset & \text{otherwise.}
\end{align*}
\]

Then \( \beta \) is an S-bound function on \( \Psi \).

Theorem 6.4 is not hard to prove; the proof appears in Appendix D. S-bound functions that obey the conditions of Theorem 6.4 are important as they are easy to generate, and in fact the bubbles or boxes that are used to circumscribe primitive objects obey these conditions (at the primitives). The power of S-bounds now follows from the fol-
lowing two theorems.

Theorem 6.5 (The Upward Theorem)

Let \( \Psi \) be a bounded tree with \( S \)-bound function \( \beta \). If \( T \) is any subtree of \( \Psi \) with immediate subtrees \( T_1 \) and \( T_2 \) then another \( S \)-bound function for \( \Psi \) is given by \( \beta' \), where

\[
\beta' = \beta[T \# S_{\beta}(T)],
\]

and the set \( S \) is given by

\[
S = \begin{cases} 
\beta(T_1 \cup S_{\beta}(T_2)) & \text{if } T = T_1 \cup T_2 \\
\beta(T_1 \land S_{\beta}(T_2)) & \text{if } T = T_1 \land T_2 \\
\beta(T_1) & \text{if } T = T_1 / T_2
\end{cases}
\]

Theorem 6.6 (The Downward Theorem)

Let \( \Psi \) be a bounded tree with \( S \)-bound function \( \beta \). If \( T \) is any subtree of \( \Psi \), and \( T' \) is an immediate subtree of \( T \), then another \( S \)-bound function for \( \Psi \) is given by \( \beta' \), where

\[
\beta' = \beta[T' \# \beta(T) \land S_{\beta}(T')]
\]

The proof of these two theorems is given in appendix D. The upward theorem states that if we take the two \( S \)-bounds from the children of a subtree then we can refine the \( S \)-bound attached to that subtree. The downward theorem states that if we take an \( S \)-bound that is associated with a proper subtree then we can use that bound to refine the \( S \)-bounds attached to its children. In practice I create an initial set of \( S \)-bounds of the type that satisfies the conditions of theorem 6.4 by boxing the primitives, and I then repeatedly use the upward theorem to refine the \( S \)-bounds in the tree, starting at the primitives and working towards the root node. Then by repeated use of the downward theorem I refine those bounds, this time starting at the root node and working towards the primitives in the tree. These
three stages can be summarised in three algorithms which I express below. The operators $\circ$ and $\circ$ are used here to denote suitable composition operations corresponding to $U$ and $A$; that is, we require for all sets $A$ and $B$

$$A \circ B \geq A \cup B$$ and

$$A \circ B \geq A \cap B.$$ 

Typically the $S$-bounds are chosen to be sets of some convenient shape, such as bubbles or boxes, and then the operators $\circ$ and $\circ$ are defined to satisfy these conditions whilst preserving the shape.

```procedure initSB( tree);
    if tree is a primitive then
        attach a convenient bound that is a superset of "tree";
    else
    begin
        boundof( tree) + $\Omega$;
        initSB( leftchildof( tree) );
        initSB( rightchildof( tree) );
    end
endproc

procedure upSB( tree);
    if tree is not a primitive then
    begin
        upSB( leftchildof( tree) );
        upSB( rightchildof( tree) );
        LB + boundof( leftchildof( tree) );
        RB + boundof( rightchildof( tree) );
        switch on operator of tree
            case $U$: $CB = LB \circ RB$;
            case $A$: $CB = LB \circ RB$;
            case $/$: $CB = LB$;
```

- 85 -
endsw

boundof( tree) + boundof( tree) \odot CB;
end
endproc

procedure downSB( tree);
if tree is not a root node then
  boundof( tree) +
    boundof( tree) \odot boundof( parentof( tree));
if tree is not a primitive then
begin
  downSB( leftchildof( tree) );
  downSB( rightchildof( tree) );
end
endproc

Then given these algorithms we can define a simple algorithm that will apply initSB() to plant the initial set of S-bounds, and then apply upSB() and downSB() as many times as required.

procedure plantSB( n, tree);
initSB( tree);
do n times
begin
  upSB( tree);
downSB( tree);
end
endproc

6.4 Examples

The result of running plantSB() is a CSG tree which has bounds embedded at every node of the tree. The bound that gets attached to the root node in the tree is a bound for the whole tree and can be used to get some idea of the size of the object; my experience
suggests that this bound is generally good enough for most purposes, even with \( n \), in \texttt{plantSB()}, equal to one. We can even allow "unboundable" primitives in the CSG tree, such as the primitive half-spaces available in \texttt{robmod} (e.g., \texttt{phalfz}, which represents \([\{x,y,z\} | z < 0\}\)), as long as they are bounded by other parts of the tree. As an example of this algorithm in action consider figure 6.4, which shows some output from \texttt{robmod}. Figure 6.4a shows a shape that is defined in \texttt{ROBMOD} as a set combination of eighteen primitive shapes, including four infinite primitives; figure 6.4b shows the boxes that are constructed as the initial S-bounds around the finite primitives, and figure 6.4c shows the bounds after a single upward and downward refining stage. Some improvement is obvious in the bounds shown, and remember that all of the infinite primitives have also had finite S-bounds placed on them. The details of the combination operators used here are given in the next section.

Figure 6.4

However, S-bounds are particularly useful when applied to intersection problems. Appendix B discusses my hidden-line removal algorithm for \texttt{robmod}, and in the notes at the end of that appendix an example of a two-dimensional intersection problem is given that is
not handled particularly well by the normal tree simplification algorithm. This example is reproduced in figure 6.5, and the idea is to produce a simplified tree that describes the rectangle \( R \) and is valid within the thin prism; the normal tree simplification rules will not remove any of the four half-planes \( \{A, B, C, D\} \) whose set intersection make up the rectangle because they all intersect the prism. We can now see how we can attach "sensible" bounds to the half-space primitives. \( R \) is represented by the tree \( AAB\overline{A}C\overline{A}D \). I have shown a box, \( X \), that is a suitable bound for \( R \). So under the relevant interpretation

\[
I(R) = I(XA) = I(XA(AAB\overline{A}C\overline{A}D)) = I((XA)A\overline{A}C\overline{A}D).
\]

So \( X \) is a suitable bound for \( A \) in \( R \), and similarly for \( B, C, D \). So we can conceptually replace the infinite half-planes by the subtrees \( XA, XA, XA, XA \). In particular this means that the simplification routine can now ignore \( A \) and \( C \) in the prism shown.

I should now point out that if we allow the cross-section of the prism to get very small then the prism looks like a 'thick' ray. So we can add a suitable call of a simplification routine to Ray_cast() (section 4.3) to take advantage of S-bounds planted in the CSG tree.
More examples of the use of S-bounds will be found in the next chapter, as they are the mainstay of my null object detection algorithm.

6.5 Boxes and Bubbles

An S-bound scheme places a suitable bound on every subtree in a tree, and this is particularly useful for algorithms that "walk" around CSG trees. The composition of two boxes (i.e., rectilinear blocks) or bubbles (i.e., spheres) can computed in a constant time, and so the whole tree can have either attached in $O(n)$ time. It is easy to see how we compute the composite boxes – each box is defined by a pair of coordinates, one giving the minimum coordinate values, and the other the maximum coordinate values. To compute the composition corresponding to the union of the boxes we just take the maximum of the maximums and the minimum of the minimums; for the intersection, we take the maximums of the minimums and the minimums of the maximums. Null bounds can, of course, be easily detected.

Compositions of bubbles are less obvious, but quite straightforward. A bubble may be denoted by its centre and its radius, so say we have two bubbles that we want to compose, denoted by $(x, r)$ and $(y, s)$ say. Without loss of generality, assume that $r \geq s$, and let $D = d(x, y)$, the Euclidean distance between $x$ and $y$. Then we have two cases and five sub-cases to consider.

**Composition is Intersection**

(1) If $D^2 > (r+s)^2$, then the result is null.

(2) If $(r+s)^2 > D^2 > r^2 - s^2$ then the bubbles overlap, and the result is a new bubble described by $(c, R)$, where
\[ R^2 = \frac{4r^2s^2-(b^2-r^2-s^2)^2}{4D^2} \quad c = x + \sqrt{\frac{r^2-R^2}{D^2}(y-x)} \]

(See figure 6.6).

(3) Otherwise, the composite is the same as the smaller bubble.

[Diagram of two overlapping circles, annotated as Figure 6.6]

Composition is Union

(1) If \( b^2 \leq (r-s)^2 \) then the composite is the same as the larger bubble, as the larger bubble envelopes the smaller.

(2) Otherwise, we have a new bubble, \((c, R)\), defined by
\[ R = \frac{(D+r+s)}{2} \quad c = \frac{1}{2} \left[ x+y + \frac{(s-r)(y-x)}{p} \right] \]

(see figure 6.7).

Note that whereas the box composition operators are associative, the bubble composition operators are not, as demonstrated by the counter examples in figure 6.8. It may also be of interest to note that the operators are not distributive either - see figure 6.9.
Figure 6.9
7. Null Object Detection

In chapter 3 I discussed the three basic clash detection techniques that I would study, and remarked that all three were dependent on being able to decide whether two entities intersected; in the multiple interference detection and sweeping methods, these entities were three-dimensional, and in the four-dimensional interference detection method the entities were four-dimensional. In turn if we represent the entities by sets, say A and B, then we note that they interfere iff their set intersection, \( A \cap B \), is non-null. This chapter is devoted to studying the null object detection problem, both in the general case but also considering the implications in the important special case of the head node of the tree being an intersection operator, which occurs when we use null object detection to perform interference detection; I shall also assume that there will often be no interference in this case. We shall see how it is easy to produce a simple solution to the problem, but that it is more difficult to produce fast solutions. Techniques that provide faster solutions are described which draw on my description of basic geometric algorithms (from Chapter 4) and the S-bound algebra (from Chapter 6).

7.1 Basic Solution

Most existing geometric modelling systems deal with NOD by trying to draw the object, and the result for a null object is either a blank picture, or even an "error" message. This sort of algorithm can be tidied up to produce a reasonable "first pass" at a NOD algorithm; I present below a two-dimensional version which is copied from [Tilove 81].

```plaintext
Boolean procedure baseNOD( T)
  foreach primitive \( P_i \) of T do
    foreach edge E of \( P_i \) do
      begin
```
Classify E wrt T,
and if any part of E is not "outside",
return false;
end

return true;
endproc

This algorithm is based on one that finds a two-dimensional wireframe; it generates a set of potential edge segments of the object, and then classifies the segments. For the NOD algorithm above we simply return false when an edge segment is found to be "inside" or "on" the object, whereas for a wireframe we just output the "on" segments.

The algorithm presented is $O(N^3)$ in the number of primitives; a three-dimensional version is $O(N^4)$, as we then have to take pairs of faces to generate candidate edges, which can then be split into potential edge segments. This algorithm can be improved slightly in domain-dependent ways, but I shall take baseNOD() as a basis for my other algorithms. For clarity I shall again concentrate on the two-dimensional case, but the ideas easily extend to higher dimensions, and I shall illustrate the ideas with examples from robmod. For consistency with what follows I shall specialise baseNOD() in a simple way; I now define it to take an extra parameter, which is the window within which the tests are to be performed.

7.2 Bath_NOD()

I described in section 4.2 a procedure, Bath_draw(), which draws a view of an object through a window by a divide-and-conquer technique. NOD may be done in the same manner, and without further ado I shall present an algorithm that uses this technique.

Boolean procedure Bath_NOD( window, tree)
    ntree = simplify( window, tree);
    if cando( ntree) then
Note that I have arranged to terminate all calculation once any call to baseNOD() returns false; that is, once we have proof of non-negativity, we can stop looking. But otherwise the correctness of the routine is just as for Bath_draw(); the complexity of this will be discussed in chapter 8. In practice I have used Bath_NOD() as it stands as the second-pass at a NOD routine, where I can supply the initial window to the system using the bounding routine on CSG trees discussed in chapter 6.

7.3 Tilove's Redundancy Routine

In [Tilove 81] a NOD algorithm is presented based on a concept that he calls "redundancy". I shall briefly describe its operation below, and then I shall describe my improved algorithm which is based on the S-bound algebra of Chapter 6.

Consider the simple arrangement of four two-dimensional primitives in figure 7.1, which illustrates a part, made from the union of the primitives A and B, touching another part, which is CUD, and we want to test for any intersection between these parts. Then we want to answer the question,

"Is \((A\cup B)\cap (CUD)\) null?"
Figure 7.1

Before I describe how the algorithm tackles this, I need to define some terminology. I define the order of a primitive $P$ in a tree $T$ to be the number of times that the path from the root node of $T$ to $P$ itself passes to the right of a difference ("/"\) operator. Informally, the order of a subtree is the number of times that it is subtracted in the tree. Then we say that primitive $P$ has positive sign \(\text{sign}(P)\) if it is of even-order, and that it has negative sign otherwise. Informally, positive primitives contribute matter to the object, whereas negative primitives take matter away.

I paraphrase Tilove's algorithm as follows:

"Given a tree, $T$, let $P$ be any positive primitive of $T$. Check whether $I(T)\text{AI}(P)$ is null, and if it is not, return FALSE. Otherwise, replace $P$ by $\bot$ in $T$, and apply the algorithm again".

This algorithm certainly works, and it is generally faster than baseNOD(\) because we are, firstly, able to just deal with $T$ within $P$, which is generally "small", and secondly, we are able to incrementally simplify $T$ after each $P$ is selected. In fact, once all the positive primitives have been dealt with in this way, we must be left with the tree that represents the null object $\bot$.\)
For the example of figure 7.1, the algorithm may proceed as follows.

1) Take $P=A$. Check that $I(T) A I(A)$ is null, and substitute to get
$I( T[A] ) = I( U B ) A ( C U D ) = I( B A ) A ( C U D )$.

2) Take $P=B$. Check that $I(B) A I(B)$ is null, and substitute to get
$I( T[A] ) = I( A ) C U D ) = 0$.
So return TRUE.

### 7.4 Recasting Tilove's Algorithm

Instead of describing Tilove's proof of the correctness of his algorithm, which is cast in terms of sets and redundancy, I shall describe a proof which follows from the results of Chapter 6.

**Theorem 7.1**

Let $\Psi$ be a tree with a totally consistent bounding function $\beta$. If $T$ is any positive subtree of $\Psi$ such that

$I(\Psi) A \beta(T) = 0$,

then $\beta(T) \Psi$ is another totally consistent bounding function on $\Psi$.

**Definition 7.2**

Given a tree, $T$, and an interpretation, $I$, then a disjoint set of subtrees of $T$, $\{T_1, T_2, \ldots, T_n\}$, is called a covering set (of $T$) if

$I( T[T_1;T_2;\ldots;T_n] ) = 0$. 

---

- 97 -
If all the $T_i$ are primitives, the set is called a primitive covering set.

By a disjoint set of subtrees we mean that no $T_i$ is a subtree of $T_j$ ($i \neq j$). Disjointness is specified because then the order of the substitutions is unimportant. The next two theorems tell us how to generate some covering sets.

**Theorem 7.3**

if $\{T_1, T_2, \ldots, T_n\}$ is a covering set for a tree $T$, so is any superset formed by adding positive disjoint subtrees.

**Theorem 7.4**

The set of the interpretations of all zero-order primitives of a tree $T$ is a covering set for $T$.

The proof of these two theorems is to be found in Appendix D. We can now show that Tilove's algorithm is correct as a corollary to these theorems. Consider the S-bound function, $\beta$, given by

\[
\beta(P) = I(P) \quad \text{if } P \text{ is a primitive of the tree},
\]

\[
\beta(T) = \overline{1} \quad \text{if } T \text{ is any other subtree}.
\]

Tilove used the set of positive primitives as a covering set; it is a covering set because it is a superset of the set of zero-order primitives. So by repeatedly applying theorem 7.1 with the S-bound function $\beta$ we either get the answer FALSE, or we eventually simplify $T$ to $\overline{1}$, and get the answer TRUE.

Theorems 7.1 and the concept of a covering set now show us that we can vary Tilove's algorithm on two counts; firstly, we can use any totally consistent bounds for subtrees in the main trees, and secondly, we can use any covering set composed of zero-order subtrees. But first I shall write out the algorithm that we now have in a pidgin-Algol format.
Boolean procedure S_NOD( tree)
NOD_bound( tree); /* see note 1) below */
T := Simplify( tree); /* see note 2) */
C := coverset( T); /* see note 3) */

while C=[]{ and T is not the null-tree do 
begin
  T' := headof( C);
  /* take the top subtree off of the list */
  if not Bath_NOD( boundof(T'), T) then
    return false;
  T := T[ T' ];
  T := Simplify( T);
end

return true;
endproc

Notes
1) NOD_bound() attaches bounds to all the subtrees in tree.
2) Simplify() replaces null-bounded subtrees by ⌘, and then uses sim¬
   plification rules like those for simplify().
3) coverset( T) returns a list consisting of a covering set of posi¬
   tive subtrees for T.

NOD_bound() has to provide a suitable totally consistent bounding
set for the CSG tree, and a suitable algorithm to do this just calls
plant SB() from chapter 6; coverset() will be detailed in section 7.7.

7.5 A Real Example

I will now illustrate the process that I have been describing by
using a real example from robmod. A Unimation Puma† 560 robot is
modelled in robmod and it is placed in a world consisting of a large
†. registered trade mark.
table, and a block; both the robot and the block are placed on the table. The system is told to look for interferences between two objects; one is the robot, and the other is the union of the table and the block. Initially the robot is as in figure 7.2; in the next few figures the modeller shows, with dotted outlines, the primitives that comprise the models, and superimposes various bounds as solid boxes on the drawing. The bounds are chosen to be boxes, orthogonal to the world axes, as described in section 6.5. Figure 7.3 shows the initial bounds around all of the primitives; there are 15 finite primitives in the model, and 4 infinite ones. Note that the infinite primitives will be finitely bounded by plantSB() at the next stage, and that for this model, all the primitives happen to be zero-order in the main tree. Figure 7.4 shows the bounds after calling plantSB(1,);† the number of bounds have been substantially reduced, but note the presence of "ghost" bounds around the main body of the robot, and around the motor casing (the single cylinder next to the robot). Using plantSB(2,) leaves a much smaller set of bounds, as shown in figure 7.5, and this set is not improved by further calls to

† i.e. after calling plantSB() with the first argument set to 1.
upSB() and downSB().
Figure 7.5

If we now turn the robot a few degrees about its waist, so that it encroaches slightly on the block, we get the set of drawings in figure 7.6. 7.6a shows the primitives plus the actual intersection volume (solid); figures 7.6b, 7.6c and 7.6d show the results of applying plantSB(1,), plantSB(2,) and plantSB(3,). Only a slight reduction is noticeable in the bound size between 7.6c and 7.6d, and I have encoded the default version of NOD_bound() in robmod to act as follows:–

```plaintext
procedure NOD_bound( tree)
    plantSB(2, tree);
endproc
```

7.6 Better Coverings

The examples above use a covering set consisting of all the zero-order primitives; we can generally find smaller covering sets than that. Consider the following (non-deterministic) procedure:
List procedure coverset0( tree)
/* coverset0 returns a list of primitives */
if Primitive( tree) then return { tree};
else
    begin
        L + coverset0( leftchild( tree));
        R + coverset0( rightchild( tree));
        on operator( tree) switch
            case U:  V + mergelist( L, R);
            case A:  V + either L or R;
            case /:  V + L;

This algorithm returns a list which is a primitive covering set for a tree; the non-determinism comes from the choice expression, "either...or", as we only need to consider a covering set for one subtree in any tree which has a leading intersection operator. In particular this happens when we use NOD for interference detection.

To implement such a strategy we need to make decisions regarding the choice of which subtree's bounds to take in the "either...or" clause, and the basic rule is to take the bounds that will result in a faster run of the algorithm. This is easier said than done though; the procedure of creating the bounds and choosing the covering set is meant to be a preprocessing stage that will improve the efficiency of S_NOD(), and we do not want to spend so much time preprocessing that it dominates the time for the entire procedure. So far the processing that we have described in S_NOD() is O(N), where N is the number of primitives in the input tree. We can preserve this behaviour by using some simple heuristics to estimate the computational cost of choosing a particular subtree at any point, and the computational gain to be had by doing so. This gives us the general form of cover-set(), as used in S_NOD(), as follows.

List procedure coverset( tree)

if Primitive( tree) then
begin
  attach cost and gain values to tree;
  return tree;
end
else
begin
  L := coverset( leftchild( tree));
R = coverset(rightchild(tree));
calculate cost and gain values from those of children,
and attach these to tree.
if the operator is A, choose the best subtree to follow.
set V to be the covering set for tree.

return V;
end

endproc

Such a framework limits the analysis of cost/gain to only looking at values calculated from neighbours in the tree, and so preserves the O(N) behaviour of the algorithm. This may be regarded as less than ideal, as proving the nullity of a subtree within a particular bound may affect subtrees that are distant in the tree when \( J \) is substituted. However in practice the method seems to work well.

I have tried one estimate for the "gain" of a subtree, and two estimates for the "cost". The "gain" of a primitive can be a measure of the complexity of a primitive, and I have used in robmod the number of planar half-spaces used to describe the primitives initially. Then the "gain" of a tree is the sum of the gains of its subtrees. The rationale here is that if we can replace the tree by \( J \), then at the next iteration we will have gained by reducing the complexity of the tree.

The "cost" of a tree is meant to reflect the difficulty of proving that the tree is null, and as in S_NOD() the time taken to prove nullity may depend on the order in which the primitives are tackled it is especially difficult to make a realistic estimate of this quantity. I have used estimates of "cost" which are a measure of the size of the bounds associated with a tree; the rationale behind this is that a tree with a larger bound is likely to intersect more primitives. Two estimates of size were tried. In one, the "size" of a bound was taken to be the sum of the lengths of the sides of the bound (with a box bound), and was meant to reflect the "diameter" of
the bound. In the other, the "size" of the bound was taken to be its volume. Both seemed to give similar results, and the resultant bound sets "looked" sensible; examples of the resultant bound sets are shown in figures 7.7-7.12.

There is one last point that I wish to make before I leave this algorithm. In S_NOD() we take the list returned by coverset(), and then use the primitives as members of a covering set in the order that they appear in the list. Although the ordering of the list does not affect the result of the algorithm it can affect the running time, and so I order the list by using a slight variant on that described in [Tilove 81]. This is done by associating with each member of the list a value, which is the ratio "cost"/"total gain", and ordering the list with respect to this value. Here "cost" is as defined earlier, and "total gain" is the gain of the subtree that will be eliminated if the primitive is proved redundant. This is a potentially $O(N \log N)$ step, but here $N$ is now the number of elements in that list, and in practice it is smaller than the original number of primitives in the tree.
Figures 7.7, 7.8 and 7.9 show the final set of bounds provided by the algorithm for the example of figure 7.2, and three values of the waist joint angles for the robot.

7.7 Effects of Loose Bounds

In the example I have used to illustrate the algorithm the boxes that were generated happened to fit some of the primitives very closely. Making the bounds a poorer fit can dramatically increase the size of the bounds in the covering set, and as an example figures 7.10 and 7.11 are of the same situations as figures 7.7 and 7.9, but now the bounds have been made a looser fit by rotating the entire model by 10 degrees about the x-axis of the world. Similar bounds follow by using other rotations. The resulting bound sets are obviously worse than the ones that were shown in previous figures, but this is not too serious for two reasons. Firstly, each set of bounds is still a "good" set, as each is certainly a better set than the set of the bounds of the primitives. And secondly, I believe that the original orientation is more likely than that in the contrived
Examples of figures 7.10 and 7.11. The problem in these later figures is that the table is a relatively large primitive, and gets bounded by a box that encompasses much of the rest of the model.
Figure 7.12 shows the same situation as that shown in figure 7.11, but the table was ignored when the bound set was calculated, and much smaller bounds have been produced. In most real cases we often find that large objects in the world are treated rather like datums, and would naturally be created in a modelling system so that they are roughly aligned with the axes, and so I expect that cases such as shown in figures 7.10 and 7.11 would be rare. If we do insist on dealing with such awkward cases, it is not difficult to imagine that we could orient the axes of the bounds so that they do fit the larger objects snugly. Another approach would be to treat things like tables specially, say by considering them as infinite planes to a first approximation, and setting up a trap that will detect any object near the plane.

† If spherical bounds are used then the same problem does not apply.
7.8 Summary

I have shown in this chapter that it is possible to construct a NOD algorithm that appears to give a good performance in some test cases of moderate complexity. The main feature of the algorithm is the preprocessing stage, which provides a set of small candidate bounds within which we may test a CSG description for nullity. The time complexity of the preprocessing stage is $O(n)$, where $n$ is the number of primitives in the original tree, plus a step which is $O(m \log m)$, where $m$ is the number of bounds in the covering set. These bounds are then passed into Tilove's redundancy-based algorithm, which incrementally simplifies the tree, and itself calls an algorithm, Bath_NOD(), whose complexity is estimated in chapter 8.

It is worth noting that all these stages can interact in various ways. For example, my experiments suggest that the bounding algorithm can substantially reduce the number of bounds fed to the redundancy stage of the algorithm, and so the effect of the incremental simplification produced by the latter routine is less noticeable than
would otherwise have been expected.

I can also imagine some other ways in which the preprocessing stages might be improved, but I have not tried to add these to the implementation. The reason for this is that it is presently difficult for me to assess the worth of any modification that I make to the system, because I cannot say what a "standard" NOD problem looks like. What is required is a large set of models and NOD problems that have been presented to a modelling system that is in everyday use. I shall, however, list possible modifications to the routine.

7.8.1 Use Different Bounds

I used orthogonal boxes as bounds, partly because they are easy to work with, and partly because many mechanical engineering set-ups do show a large degree of orthogonality. Other bound sets might include ellipsoids ("sausages") or non-orthogonal boxes; any types of bounds that cannot be combined in pairs in a fixed time may imply a preprocessing time greater than $O(n)$.

7.8.2 Use More Bound Refinement Steps

I settled on two sweeps up and down the CSG tree to create my bounds, but this was just a subjective choice. Any fixed number of steps will result in an $O(n)$ preprocessing algorithm, but anything more elaborate may increase this complexity.

7.8.3 Use Better Covering Set Heuristics

The heuristics that I used to pick out my covering set were crude, but seemed effective. Better heuristics might involve searching the tree and noting, for instance, which bounds do intersect each other,
as this would suggest that their contents might well intersect.
3. Statistical Analysis of Algorithms

This chapter is concerned with the statistical analysis of some of the algorithms that I have developed in this thesis. The statistical analysis of algorithms is a difficult problem to solve in general, and I only give a rigorous analysis for the algorithm Bath\_NOD() from chapter 7. The method of analysis is based on that described in [Tilove 81], but I have used a reformulation of Tilove's analysis that allows me to analyse the Bath group's tree simplification routine, and to cater for variable sized primitives in the statistical model.

The basic idea behind the statistical analysis of algorithms is quite simple. We have a cost function (or functions) which is the time complexity of an algorithm expressed as a function of some suitable parameters. We then assume that the values that the parameters take (say $\rho$) follow some statistical model - that is, if we could repeatedly take samples of the parameters by repeating the experiment (i.e., running an implementation of the algorithm), then we would expect the numbers of times that a particular set of parameters is found would, after a long time, approximate to the numbers predicted by the statistical model. Under these assumptions if our cost function is $c(\rho)$ then we say that the expected value of the time complexity is the average value of $c(\rho)$ - i.e.,

$$E[c(\rho)] = \sum\text{Probability that } \rho \text{ has a certain value } \times c(\rho)$$

if the set of values for $\rho$ is discrete; otherwise, we can replace the summation by an integral and the finite probabilities by a probability density function (pdf). These concepts are covered in most standard texts on probability theory, such as [Papoulis 65] or [Feller 68].

We can identify certain requirements that we must fulfil before we can compute these expectations. Firstly, we must have in mind a particular universe of discourse, which in our case means a certain set
of primitive shapes with which we shall construct our geometric models. Secondly, we need to formulate our cost-functions. And thirdly, we need to define the statistical model that we shall use. When we analyse algorithms in this way we do not consider these three steps in isolation, but rather we see how the three can be restricted, to make the solution simpler, or broadened, to make the results more general. It is only with hindsight that we can present these three as well-partitioned entities.

8.1 Analysing NOD

The problem that I would like to look at is NOD. Unfortunately the behaviour of the major routine of the previous chapter, S_NOD(), is not well enough understood for me to attempt a full analysis, because it is difficult to guess how well the preprocessing stages will work. Recall that these stages call Bath_NOD() with successively smaller CSG trees, and in "small" windows. The implementation of this routine (in robmod) is not old enough for me to make anything other than informed guesses about the size of the trees that are fed to Bath_NOD(), but I feel that a pair of reasonable working assumptions are that

(a) the trees that are fed to Bath_NOD() are a fraction of the size of the original tree, and

(b) the content of the "windows" fed to Bath_NOD() are not easily proved to be null.

Assumption (b) comes about because the S-bounds that are used as the "windows" for Bath_NOD() are good approximations to where any part of the object (that we are testing for nullity) may be. If Bath_NOD() is used to perform NOD by itself then one generally finds that many of the octants generated by the divide-and-conquer process are easily proved null, as their local CSG trees simplify to .
What these comments suggest is that the computational complexity of Bath_NOD(), when used as part of S_NOD(), is rather like the similar routine to compute a wireframe. This is making an implicit assumption that we expect the object to be null, and so cannot gain much from the way that the algorithm finishes as soon as non-nullity is proved. So to emphasis that I am working under these assumptions I shall call the problem that I am studying DENSE_NOD. [Tilove 81] contains a study of the similar WIREFRAME problem, and as this was the starting point for my own analysis I shall quote his informal definition below, together with a similar definition of DENSE_NOD.

WIREFRAME: Given a CSG representation of a solid, S, produce the edges of S.

DENSE_NOD: Given a CSG representation of a solid, S, confirm that its edge set is null.

I shall follow Tilove and use a special universe of discourse for studying the complexity of DENSE_NOD, which I will call the squares world. In the squares world the CSG models are of two-dimensional objects, which are made up by the usual set operations using just one type of primitive, squares, and the sides of the squares are parallel to a set of cartesian coordinate axes. Furthermore, for the first stage of the analysis I shall assume that the squares are all of the same size. As the chapter progresses I shall explain how the results extend to higher dimensions, and how we might consider a richer set of primitives. But first I shall discuss Tilove’s algorithm for WIREFRAME, and the (statistical) model that he used to analyse it.

8.2 Tilove’s Analysis

Tilove used a universe of discourse in which the square primitives, of fixed size, could lie at random anywhere on a large square. I have changed his notation slightly in what follows, and called the resulting universe of discourse the squares world. In this the primitives are viewed through a unit square window, $[0,1] \times [0,1]$, and the
primitives themselves are also squares. The reason for fixing the size of the universe to a unit is for simplicity; it is justifiable as a modelling system may find a bound for its entire set of objects, and scale the primitives so that the bound fits into a unit cube (internally robmod does just that). In Tilove's model, the primitives are of fixed size, \( r \), and he places them at random on the world by placing their lower left-hand corner at random in the square \([0,1-r] \times [0,1-r]\) (figure 8.1). Tilove's algorithm works by fixing a grid of windows over the main window (dotted in figure 8.1), and computing the wireframe through each window. It differs from the Bath group's algorithm in that the windows are chosen once and for all at the start of the algorithm, and that only simplification steps corresponding to my \( \downarrow \)-simplification steps (cf. figure 4.5) are used to produce the simplified trees within each window. The window size is denoted by \( d \), and the number of windows by \( M \), and hence

\[
Md^2 = 1 \tag{8.1}
\]

Three cost functions are considered; the first refers to the time taken to compute the wireframe in each window (C); the second refers to the time it takes to decide which primitives intersect which window (PCI); and the third refers to the time taken to produce the simplified trees (SIMPPLY). The bracketed terms refer to the terms used in [Tilove 81].
The total time for \( C \) is given by Tilove to be bounded by a constant multiple of the following:

\[
F = \sum_{k=1}^{M} \left( E_k N_k^2 + N_k \right)
\]  

Here \( N_k \) refers to the number of primitives that intersect the \( k \)th window, and \( E_k \) to the total number of edges that appear in the \( k \)th window. (Tilove uses the term "cell" where I use "window".) The cost functions for PCI and SIMPLIFY have simpler forms — namely, if \( N \) is the number of primitives,

\[
\text{PCI is } \Theta(M \times \text{Total Area of Primitives}) \tag{8.3}
\]

\[
\text{SIMPLIFY is } \Theta(N^2) \tag{8.4}
\]

Tilove assumes that the square primitives are placed independently at random in the world, and by making a further assumption about the statistical model computes a value for (8.2). This extra assumption he calls the constant-density assumption, and it translates under my version of the squares world into the statement that the total area of the primitives is bounded over all \( N \). Then by choosing \( M \) proportional to \( N \) (abbreviated as \( M=\alpha N \)) equation (8.2) gives the complexity of \( C \) to be \( \Theta(N) \). In this case WIREFRAME is asymptotically time-bounded by the \( \Theta(N^2) \) SIMPLIFY cost, although Tilove does point out that he would expect the SIMPLIFY time to be smaller in practice as the \( \Theta(N^2) \) cost is for a worst-case input tree. Also, in practice the time to do SIMPLIFY will still be smaller than the time to do \( C \) for any presently practicable \( N \).

### 8.3 Analysis of Bath NOD()

The algorithm Bath NOD() was detailed in section 7.2. In it the (two-dimensional) windows were successively divided by four until the tree was simple enough to be tackled. The device that decided whether to divide or to conquer a particular window was the predicate cando(); I will simplify my analysis considerably by only considering in detail a version of cando() that allows division a fixed number of
times, which can be decided by the algorithm after an initial tree-scan (an $O(N)$ step), and then conquering. Let the number of times that division takes place be $D$; then the window is divided up into $M$ square windows of equal size, where

$$M = 4^D$$

As DENSE_NOD for each window is rather akin to WIREFRAME, we can see that if we use Tivoli's statistical model (constant density, and independence) then choosing $M=N$ should give us his $O(N)$ cost for the "conquer" part of the algorithm. This in fact does follow, but I shall detail the analysis below.

We are referencing the primitives by the position of their lower left-hand corner and so, for a given window size $(d)$ and a given window $(W')$ we can compute the particular subset of $W = [0,1] \times [0,1]$ that this corner must occupy for the primitive not to be simplified out of the local CSG tree for $W'$. For our square primitive this happens if

$$r < d$$

or if

$$r > d$$

Figure 8.2

any edge of the primitive is visible in $W'$. Figure 8.2 shows the subset for a $W'$ in the middle of $W$, and for the two cases $r < d$ and $r > d$. A primitive is shown hatched, $W'$ is shown solid, and the outline of the subset, $V$, is shown dotted (but includes $W'$).
If we refer to the lower left-hand corner of a particular primitive by the coordinates \((x,y)\) as shown, then our statement that the primitives are to be regarded as independent implies that the probability that the primitive is to be found in the window \(W'\) is given by

\[
\frac{\iint V 1 \, dx \, dy}{\iint_{[0,1-r] \times [0,1-r]} 1 \, dx \, dy} = \frac{\text{Area of } V}{(1-r)^2}
\]

I shall now make an approximation - namely, that the denominator is 1, and just take the value of this probability as

\[
\text{Area of } V
\]

This approximation ignores border effects - that is, what happens when the primitive is near the boundary of \(W\). I write \(A(r,d)\) for the area of \(V\), and note that

\[
A(r,d) = \begin{cases} 
(r+d)^2 & \text{if } r < d \\
4rd & \text{if } r > d 
\end{cases}
\]  

(8.5)

This was for a window \(W'\) near the centre of \(W\). We also assume that the effect of different expressions for windows near the boundary of \(W\) will have negligible effect, and take \(A(r,d)\) to be the probability that a particular primitive affects a particular window for all windows (what I call the homogeneity assumption).

I now define the cost function for the "conquer" part of \(\text{Bath}_NOD()\) as follows. We are interested in the dominant term in the calculation of the complexity of \(\text{baseNOD}()\), and it is easy to see that, using Tilove's nomenclature, the complexity for the \(k\)th window is bounded by a constant multiple of

\[
\frac{E^3}{r^k}
\]

and so the total cost is bounded by a constant multiple of
\[
M \sum_{k=1}^{E_k^3} k
\]

However, in turn if \( P_k \) is the number of primitives in the local CSG tree for the \( k \)th window we see that \( P_k \leq E_k \leq 4P_k \), and so we can consider our cost function to be given by a constant multiple of

\[
F' = \sum_{k=1}^{M} P_k^3
\]

(8.6)

Now each \( P_k \) is a random variable with distribution defined by the statistical model, and by the homogeneity assumption each \( P_k \) has an identical distribution. It also follows, by the assumption of independence, that the form of each \( P_k \) is actually of a Bernoulli trial. A classic example of a Bernoulli trial is that of an experiment in which we count the number of heads in some fixed number of tosses of a coin. In our case, we want the number of primitives affecting the cell, given that each primitive has a probability of \( p = A(r,d) \)
of affecting the cell. We can now find the expected value of \( F' \) in (8.3) by calculating the expected value of \( P_k^3 \); for a Bernoulli trial of \( N \) "attempts", each with probability \( p \) of success, it is

\[
E(P_k^3) = N(N-1)(N-2)p^3 + 3N(N-1)p^2 + Np
\]

where \( N \) is, of course, our number of primitives in the original tree for \( W \). This expression is most easily calculated by the method of generating functions. Summing this expression over all \( M \) windows gives

\[
E(F') = M[N(N-1)(N-2)p^3 + 3N(N-1)p^2 + Np]
\]

(8.7)

Now we can optimise our algorithm for the constant-density condition. Our algorithm is given a tree, and can count, say, the number of primitives in the tree, and if the algorithm assumes that the tree follows the statistical model that was detailed above it can decide on a value of \( M \) that will minimise \( E(F') \). The value to take is \( M = N - i.e., D = \log N \). Then \( d \) varies as \( O(N^{-\frac{1}{2}}) \), \( r \) is \( O(N^{-\frac{1}{2}}) \), and so \( p \) is
\(O(1/N)\), and

\[ E[F'] = M \cdot O(1) = O(N) \]

as we expected from Tilove's analysis.

For clarity in what follows I shall refer to the computation whose cost I have just bounded as the CONQUER part of Bath_NOD(); I shall refer to the rest of the computation involved as the DIVIDE part. I shall now find a bound for the cost of DIVIDE. Let \( n \) be the number of primitives in the tree that is given to a particular window after the algorithm has divided \( D' \) times \((0 \leq D' \leq D)\). Then the tree is simplified, to contain \( n' \) primitives, and this tree is given to each of four new windows. So the immediate processing of this window is an \( O(n) \) step, and it spawns four processes, whose immediate costs are \( O(n') \).

Now rather than trying to sum these costs along the tree of process calls, consider the following. A suitable cost function for the DIVIDE time complexity is given by the sum, over all calls to Bath_NOD(), of the number of primitives that it is given in the CSG tree. This sum is also given by the sum, over \( D' \) and \( 0 \leq D' < D \), of the total number of primitives in the input trees of all calls of Bath_NOD() which are the child processes of \( D'-1 \) calls to Bath_NOD(). But the total number of primitives fed to all calls of Bath_NOD() after division step \( D' \) is the sum, over all primitives, of the number of windows containing the boundary of a particular primitive. This is illustrated in figure 8.3, with three primitives \((A,B,C)\) and four windows \((1,2,3,4)\). Then

number of primitives in window 1/2/3/4 is 2/2/1/1 (total 6)

number of windows around primitive A/B/C is 1/1/4 (total 6)

So this implies that the complexity cost at a division depth \( D' \) is the sum, over all primitives, of the number of windows that each primitive effects. And the number of windows that a primitive effects is precisely the number of times that the grid, which defines
the $4^{D'}$ windows, cuts the boundary of the primitive. Furthermore, if $d>r$ then each primitive can be cut at most four times — otherwise, the lines of the grid would have to be closer together than $r$. Thus,

$$\text{If } d>r, \text{ total cost of DIVIDE at depth } D' \text{ is } O(N)$$  \hspace{1cm} (8.8)

Finally, note that for our case choosing $M\leq N$ meant that the final value of $d$ was $\Omega(r)$, and so $d>r$ for all but some bounded number of division depths. Thus the total cost of DIVIDE is

$$O(DN) + O(N) = O(N\log N)$$  \hspace{1cm} (8.9)

which is better than the cost of SIMPLIFY given in (8.4).

### 8.4 Extensions to Three Dimensions

Here I sketch the differences in the analysis of a three-dimensional cubes world. In this new universe of discourse the primitives are orthogonal cubes, whose sides are parallel to the coordinate axes and which are lying in $[0,1]^3$. Making the same assumptions as for the squares world leads us to define a region for each particular three-dimensional "window" and primitive size $r$, so that the primitive affects the window if a particular corner lies in the region. Thus we get a function $V(r)\leq(r+d)^3$, which again gives the probability $p$ in a Bernoulli trial, and so the cost function for CONQUER becomes
\[ \mathbb{E}(p_k^4) = N(N-1)(N-2)(N-3)p^4 + 6N(N-1)(N-2)p^3 + 7N(N-1)p^2 + \text{Np}. \]

Assuming constant-density implies that the total volume is bounded, and so \( r = O(1/3/N) \); choosing \( M = N \) gives \( d = O(1/3/N) \), and so the cost function becomes \( O(N) \), as before. Similarly, the cost of DIVIDE becomes \( O(N \log N) \).

This result may seem surprising; however, note that in both two- and three-dimensions the assumption of constant-density and the choice of \( M \) implies that the expected number of primitives in each window is bounded, and so each window can be solved in constant time.

### 8.5 Realism of the Statistical Model

The results above are interesting, but we would be right to ask ourselves whether they paint a realistic picture of the expected complexity of Bath_NOD when that algorithm is given an "average" tree as input. I can identify five possible problems with the analysis so far, and I list them below.

1. The shape of the primitives is very simple. Square (or cube) primitives are fairly unlikely as real primitives, but they are easy to treat. The main influence of the shape of the primitives is in deciding the shape of the region \( V \) (figure 8.2), and the area of \( V \) would be roughly equivalent to our squares case for "roughly" square primitives. Primitives that have a prominent major axis might significantly deviate from this norm; if such primitives are thought likely, we can either extend our analysis, or we treat them as the union of a number of roughly square primitives.

2. The primitives are placed randomly. If we had been analysing WIREFRAME then this assumption would be suspect, as we expect all of the primitives to be well within \( W \), and so they are less likely to be near the boundary of \( W \) than near its centre. This is less likely to be true for DENSE_NOD, as we then have our \( W \) as a window on just a subset of the total number of primitives (e.g. figures 7.7-7.9). In
either case, this assumption is seen only as an approximation, just as our assumption of homogeneity is another approximation. I regard it as a reasonable approximation (but see note (5) below).

(3) The primitives are of single size. This is rarely a feature of primitives in a real CSG tree. However it is possible to make a further analysis in which the primitives have a random size, and this will be detailed in the next section.

(4) We used the constant density model. This assumption seemed suspect to me when I first encountered it, but on further reflection I feel that it is justifiable, at least as a first approximation. A CSG model of an object is often created by first making a rough model, and then adding successive layers of detail by using smaller primitives which are placed near the surface of the model. If more primitives and more detail are added to a model, then either the total size of the model goes up, and so the relative size of the primitives goes down, or the size of the new primitives is small, and they make little contribution to the volume of the model. This is, of course, a generalisation, and one should be wary of any analysis that is that is particularly sensitive to this assumption.

(5) We assumed that the primitives were placed independently. This assumption is certainly not true, as primitives are invariably arranged in a well-ordered fashion, with planes parallel, shafts coaxial, etc. So one cannot justify this assumption, but I can 'wave my hands' to try to estimate the effects that it has on my analysis. Dependencies like having planes parallel, but far apart, should have a fairly small effect, but coincident surfaces could cause problems, as they will be 'hot-spots' of complexity. However this effect may be minimised by recognising such hot-spots as 'special cases' in the modelling system. For example, robmod identifies all coincident faces in a preprocessing stage, and so can treat such faces as one face in complexity counts (although they will still take up two or more CSG tree nodes). Such special-case treatment may help to reduce the asymptotic costs of such routines (with respect to the size of
the CSG tree that is input).

8.6 An Analysis Incorporating Variable Sized Primitives

I shall now extend my analysis of Bath NOD( ) to allow variable sized primitives. To do this I will use the same universe of discourse (the squares world), and the same cost functions (for DIVIDE and CONQUER), but now the statistical model will be altered slightly. I will still assume that my primitives are placed independently in the world, and also that the probability that a primitive affects a window is still given by (8.5), but now I assume that the size of each primitive is drawn from a probability distribution — namely, a lognormal distribution. This distribution was chosen partly because it seemed to match the distribution found in 'real' models, and partly because it is amenable to manipulation.

More precisely, I have denoted the size of the primitives by \( r \); so I let \( r \) be given by an equation

\[
    r = e^{\mu + \sigma x}
\]  

(8.10)

where \( x \) is drawn from a unit normal distribution; that is, \( \log r \) has a Gaussian distribution.\(^\dagger\) So if I use \( \phi() \) to denote the cumulative normal distribution function, then

\[
    \phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}v^2} \, dv
\]  

(8.11)

and the properties of \( \phi() \) include

\[
    \phi(-\infty) = 0 \quad \phi(\infty) = 1 \quad \phi(-x) = 1 - \phi(x)
\]  

(8.12)

I can now estimate bounds for the costs of CONQUER and DIVIDE. For CONQUER I have to find the expected value of (8.5), and this will give us \( p \), the probability that any particular primitive will be

\(^\dagger\) Note that any primitives that are larger than the initial window can generally be simplified to something of roughly unit size. Thus I shall assume that \( \mu < 0 \).
found in a given window. Then as I have again assumed independence I again have the situation of a Bernoulli trial, and can use (8.7) to estimate the cost of CONQUER. The expected value of \( A(r,d) \) is given by

\[
p = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} A(r,d) e^{-\frac{1}{2}v^2} dv \quad \text{where} \quad r = e^{\mu + \sigma v}.
\]

\( \mu \) and \( \sigma \) are commonly called the mean and standard deviation of the underlying normal distribution, and I shall make some assumptions as to their range of values later. So if we let \( \alpha \) be the value of \( x \) when \( r=d \) then

\[
\alpha = \frac{1}{\sigma} \log(d/r) \quad \text{(8.13)}
\]

from (8.10), where \( r = e^{\mu} \), and so (using (8.5))

\[
p = \frac{1}{\sqrt{2\pi} \sigma} \int_{-\infty}^{\alpha} (r+d)^2 e^{-\frac{1}{2}v^2} dv + \frac{1}{\sqrt{2\pi} \sigma} \int_{\alpha}^{\infty} 4rde^{-\frac{1}{2}v^2} dv
\]

If \( r<d \) then \( (r+d)<2d \), and therefore

\[
p < \frac{1}{\sqrt{2\pi} \sigma} \int_{-\infty}^{\alpha} 4d^2 e^{-\frac{1}{2}v^2} dv + \frac{1}{\sqrt{2\pi} \sigma} \int_{\alpha}^{\infty} 4rde^{-\frac{1}{2}v^2} dv
\]

This gives

\[
p < 4d^2 \Phi(\alpha) + 4dr \Phi(\alpha-\sigma) \quad \text{(8.14)}
\]

as

\[
\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{X} e^{\mu+\sigma v} e^{-\frac{1}{2}v^2} dv = r e^{\frac{1}{2} \sigma^2} \Phi(x-\sigma) \quad \text{(8.15)}
\]

from lemma E.1 (appendix E), and (8.12).

This gives us \( p \), but before we calculate the cost for CONQUER I shall analyse DIVIDE. Recall that the cost of DIVIDE is the sum, over all primitives and calls of Bath_NOD(), of the total number of edge segments that each division grid creates from the boundary of each primitive. We have three cases to consider:
a) if \( r > 1 \) then the boundary of the primitive is split at every depth of division. So the number of segments is

\[
\Theta(1+2+4+\ldots+2^D) = \Theta(2^D) = \Theta(1/d)
\]

b) if \( 1 > r > d \) then the boundary of the primitive is split only when the grid size of the subdivisions becomes \( \Theta(r) \). So the number of segments is

\[
\Theta(-\log r + r/d)
\]

c) if \( d > r \) then the boundary is split either once or not at all. So the number of segments is

\[
\Theta(-\log d)
\]

Collating these three cases gives the number of segments to be \( \Theta(S(r)) \), where

\[
S(r) = \begin{cases} 
-\log d & \text{if } d > r \\
-\log r + r/d & \text{if } 1 > r > d \\
1/d & \text{if } r > 1 
\end{cases}
\quad (8.16)
\]

Now if \( E(S) \) is the expected value of \( S(r) \) then

\[
E(S) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\alpha} (-\log d) e^{-\frac{1}{2}v^2} dv + \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\mu/\sigma} (-\log r + \frac{r}{d} e^{-\frac{1}{2}v^2}) dv
\]

\[
+ \frac{1}{\sqrt{2\pi}} \int_{-\mu/\sigma}^{\infty} e^{-\frac{1}{2}v^2} dv
\]

\[
< \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\alpha} (-\log d) e^{-\frac{1}{2}v^2} dv + \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\mu/\sigma} e^{-\frac{1}{2}v^2} dv
\]

\[
+ \frac{1}{\sqrt{2\pi}} \int_{-\mu/\sigma}^{\infty} e^{-\frac{1}{2}v^2} dv
\]

as \( -\log r < -\log d \) for \( 1 > r > d \). So

\[
E(S) < (-\log d) \phi(-\mu/\sigma) + \frac{r}{d} e^{\frac{1}{2} \sigma^2} \left[ \phi(-\mu/\sigma) - \phi(\alpha-\sigma) \right] + \frac{1}{d} \phi(\mu/\sigma) \quad (8.17)
\]

using (8.12) and (8.15)
Finally, we can calculate the expected total area. The expected area of a primitive is $E(r^2)$, or

$$
\frac{1}{\sqrt{2\pi}} \int e^{2\mu+2\sigma v} e^{-\frac{1}{2}v^2} dv = (r_\mu e_{\sigma^2}^2)
$$

by lemma E.2 (appendix E), and so the total expected area is given by

$$\text{Total Area is } \left[ N(r_\mu e_{\sigma^2}^2) \right]^2 \tag{8.18}$$

So if we assume constant density again, then the value of (8.18) must be bounded, and

$$r_\mu e_{\sigma^2}^2 \text{ is } O(N^{-\frac{1}{2}}) \tag{8.19}$$

Now examining (8.12), (8.14) and (8.19), then we have $q(x) \leq 1$ for all $x$, and $r_\mu e_{\sigma^2}^2 \text{ is } O(N^{-\frac{1}{2}})$ (as it is less than $r_\mu e_{\sigma^2}^2$), and so

$$p \text{ is } O(4d^2 + 4dN^{-\frac{1}{2}}) \tag{8.20}$$

So if we chose $M=N$ as before, then $d \text{ is } O(N^{-\frac{1}{2}})$, and so $p$ will be $O(1/N)$, and by (8.7),

$$\text{Cost of Conquer is } O(N) \tag{8.21}$$

as for the fixed size model. However the situation for DIVIDE is not so clear. The first two terms of (8.17) become bounded by $O(\log N)$ and $O(1)$ respectively. The third term can be naively bounded by $O(\sqrt{N})$, but then this term would dominate our expression, and we would have the result

Constant Density Only:

$$\text{Cost of DIVIDE is } O(N\sqrt{N}) \tag{8.22}$$

We can restore the $O(N \log N)$ cost if we are a little more restrictive. A sufficient set of restrictions are that $q$ is bounded, and then that $r_\mu$ is $O(d)$. Then if we consider the third term of (8.17), and note that
\[
\phi(\mu/o) < \frac{1}{\mu/o\sqrt{2\pi}} e^{-\frac{1}{2}(\mu/o)^2}
\]

by lemma E.3 (appendix E), \(d = O(r_m) = O(e^H)\), and \(\lim_{N \to \infty} \mu = -\infty\), and so

\[
\frac{1}{d} \phi(\mu/o) \text{ is bounded by a constant multiple of}
\]

\[
\frac{1}{e^H} e^{-\frac{1}{2}(\mu/o)^2} = e^{-\frac{1}{2}(\mu/o + o)^2} e^{\frac{1}{2}o^2} \ll e^{\frac{1}{2}o^2}
\]

which is itself bounded. So we have the result

Constant Density and Well-Behaved \(o\):

Cost of DIVIDE is \(O(N \log N)\) \(\quad (8.23)\)

8.7 Implications of this Analysis

In the last section I have derived some bounds for the complexity of the operations in Bath_NOD() under a certain statistical model which allows primitives of random size. I have shown that the cost of the simplified form of Bath_NOD() is dominated by the DIVIDE cost, which is \(O(N/N)\) under the constant-density model, or \(O(N \log N)\) if we are a little more restrictive. I have carried out a further analysis of these expressions, but I have not found the results useful.

It is now time to take a step back from the mathematics. The advantage of carrying out the analysis has been that we now have some feel for the effects of primitives of different sizes in our model. The results are encouraging because they show that the estimated complexity of Bath_NOD() is not dramatically degraded in our new model. The analysis also raises further questions, like "What are the typical values of \(\mu\) and \(o\), and how do they vary with \(N\)"? An indication of the variation in \(o\) for real models should be given roughly by the \((\log\text{ of the})\) ratio between the largest and smallest primitives, and so should be a measure of the detail in a model. I suspect that this is not likely to increase with \(N\), or at least, not dramatically, and the conditions quoted in deriving (8.23) could be relaxed a little for that result still to hold. A full analysis has not been carried
out, but I conjecture that taking $\sigma$ as being $O(\log\log N)$ will not cause problems, and under the constant density model this will be sufficient to bound $\mu$ too. The constant-density assumption could also be relaxed slightly without adversely affecting the complexity bound.

What I have started to do is to collect empirical evidence for the suitability, or otherwise, of this statistical model. Originally the lognormal distribution was chosen because the distribution seen in real models "looked" like this distribution (using robmod). Now robmod has been equipped with a function, analyse, which takes a shape and produces some relevant statistics. The function measures the size of each primitive by taking the log of the cube root of its volume. It then displays the list of such sizes, together with some parameters, which are the estimated parameters of the distribution. It also includes a measure of the total volume of all the primitives, expressed as a ratio of the volume of the total bound on the tree, and the ratio of the value of $(8.18)$ to the total volume, using the estimated parameters. An example of the output of analyse is shown below.

Shape analysis: 18 primitives, 4 infinite

once bounded and scaled -
volume range 0.000 to 0.061; size range 0.071 to 0.394
log[size]
-2.640, -2.640, -2.453, -2.453, -1.576, -1.350,
-1.340, -1.332, -1.266, -1.266, -1.144, -1.061,
-0.949, -0.949, -0.932, -0.932, -0.932, -0.932
estimated parameters (total volume is 0.544)
mu sigma2 skew exp(sigma2)/n est vol/vol
-1.453 0.399 -0.627 0.083 4.027

It is still too early to say whether the objects that are passed to analyse do seem to follow the lognormal distribution, although I have seen nothing yet to suggest the contrary. A more satisfactory arrangement for collecting the data would be to arrange for the users
of a modelling system to be able to pass on interesting objects for analysis, but there are not yet enough people using robmod to make that worthwhile.
9. Multiple Interference Detection

This chapter is concerned with the first of the three methods that I will consider for detecting clashes, namely Multiple Interference Detection. Multiple interference detection is conceptually the easiest of the three methods to understand, for if two objects clash then they must interfere at some point in time, and if they interfere at some point in time then they clash. So if we can do interference detection at a sufficient number of time values we must be able to detect whether objects clash. The problem, as was mentioned in Chapter 3, is to choose the time values in a sensible manner.

In this chapter two "intelligent" algorithms for choosing the times are discussed. Both of these algorithms rely on the calculation of the minimum distance between objects, and on the estimation of the relative speeds between objects. The calculation of minimum distance turns out to be straightforward, but slow, and my standard computational techniques from Ch.4 are not very helpful. Instead I describe some heuristics that have been employed to speed up this calculation. I conclude this chapter with some examples which compare these algorithms against two simpler algorithms, all of which have been implemented in ROBMOD.

9.1 Achilles and the Tortoise

I have dubbed the first algorithm that I will consider "Achilles and the Tortoise" for reasons which should become clear. The algorithm is applicable over a time interval, say \([t_s, t_f]\), and at any stage in the algorithm there is a state variable that represents the current time, \(t \in [t_s, t_f]\). The algorithm tests for interference between the objects, and if there is none it decides how far ahead in time it can safely jump and still be sure of not missing a clash. Eventually either an interference is found or \(t\) becomes greater than \(t_f\), and in either case the algorithm terminates.
The algorithm must be able to access a procedure that will estimate a bound on the relative speed of a pair of objects in the time interval \([t, t_f]\). Further it can calculate the distance apart of the objects at time \(t\), and thus if it chooses a time-step of minimum distance / maximum speed then it cannot miss a clash. A pidgin-Algol version of the simplest variant on this algorithm is given below.

```algol
Boolean procedure ClashAT(t_s, t_f, obj1, obj2)
    finished ← FALSE;  t ← t_s;
    while not finished and t < t_f
        begin md ← mindist(obj1, obj2, t);
            if md ≤ 0 then
                finished ← TRUE;
            else
                t ← t + (md / maxspeed(obj1, obj2, t, t_f));
            end
        return (t < t_f);
    endproc
```

`maxspeed()` need not give an accurate value for the maximum relative speed of the objects. In fact for the examples shown in this thesis a trivial version of `maxspeed()` was used which returned a constant value. This version was used for two reasons; firstly because it was sufficient for the examples shown; and secondly because in practice the implementation of `maxspeed()` would be strongly dependent on the domain of interest. As an example we might implement this algorithm for use by a robot dynamics simulation program, such as the program `sim` ([Featherstone 82]), and then we would expect the robot dynamics simulator to provide specialised information for `maxspeed()`.

Unlike the estimation of maximum speed the problem of calculating the minimum distance between objects is a purely geometrical problem and can be answered solely by a geometric modelling system.
algorithm for calculating minimum distance will be detailed in sections 9.3 and 9.4.

As an example of this clash detection algorithm in action consider the robot workshop model used to illustrate Chapter 7. Figure 9.1 is a wireframe picture of a Unimation Puma 600 robot set up on a table on which there is also a simple obstacle. Figure 9.2 then shows a set of positions of the lower arm assembly of the robot whilst it is performing a smooth motion. The motion considered is generated by driving each of the upper three joints of the robot at a constant angular speed, and the positions shown are those at which my implementation of ClashAT() performs a static test. As can be seen no clash occurs during this particular motion.

One problem with ClashAT() as shown is that it may never terminate if the objects do clash. This is because the time-step is calculated to ensure that the objects can never come into contact, and so unless the objects approach each other at their maximum relative speed the time steps will decrease, but no interference will ever be found. In practice we can work around this problem in a number of ways. One
solution is to relax the insistence on always being able to find any clash that occurs and only look for any clash that lasts for a time greater than some small $\Delta t$. This is done by replacing the command line from ClashAT() that calculates the time-step by

$$t + t + \Delta t + \left( \frac{md}{\text{maxspeed}(obj1, obj2, t, t_f)} \right);$$

and this is sufficient to ensure the finite termination of the algorithm. Another approach is to only detect clashes that cause a mutual interpenetration of the objects by at least $\Delta d$, and this is done by replacing the same command line in ClashAT() by

$$t + t + \left( \frac{md + \Delta d}{\text{maxspeed}(obj1, obj2, t, t_f)} \right);$$

If as well as using this second modification we also use models of the objects that are slightly larger than the real objects then we can be assured of always finding any clash. To do this we must grow a "skin" around all of the models with a thickness of at least $\Delta d$. Of course there is now the possibility that we might report a clash between these enlarged models that does not really happen between the real objects.
In practice I suspect that it might often be preferable to use a hybrid algorithm to solve the clash detection problem in these cases. Such an algorithm might use ClashAT() to get near any potential clashes in both space and time, and then use one of the other techniques to finally check for a clash. This will be discussed in more detail in Chapter 11, but to show that we can normally rely on ClashAT() to get as close as we please to a clash I will state and prove a weak form of convergence for this algorithm.

**Theorem 9.1**

Assume that at some time two objects are a distance $d$ apart and we estimate that they have a maximum relative speed of $s$ towards each other. If in their subsequent motion their true relative speed towards each other is bounded below by $\lambda s$ ($\lambda > 0$) then the algorithm described will discover that the objects approach to within a distance $\mu d$ of each other ($\mu > 0$) in at most $m$ steps, where

$$(1-\lambda)^m = \mu$$

**Proof**

The first time step chosen by the algorithm is $d/s$, and so the smallest amount by which the minimum distance will be reduced is $\lambda sd/s = \lambda d$. So the distance between the objects will be at most $(1-\lambda)d$ after one iteration, and similarly it follows that after $m$ iterations it will be at most $(1-\lambda)^m d$.

QED

Thus for many motions the algorithm will get as close as we please to a clash. One significant class of objects and motions where the algorithm does fail are those involving a sliding contact. For example figure 9.3 is a side-view of a rectangular block sliding on an L-shaped block. There is never any interference between the lower part of the L-shaped block and the rectangular block, but the minimum distance between the two blocks is zero. The only solution would be
to use some expert that could ignore the contact occurring between the lower part of the L-shaped block and the rectangular block. Such an expert might be a human or it might be a computer program.

Figure 9.3

A further example of the behaviour of ClashAT() will be given in section 9.5 when the various forms of multiple interference detection discussed here will be compared.

9.2 Divide and Conquer

"Achilles and the tortoise" used knowledge about the speed and positions of a pair of objects at a particular time to guide its behaviour. In contrast the algorithm described in this section considers a whole time-span and bases its behaviour on the state of the world at both extrema of the time-span. This new algorithm is a divide and conquer algorithm. Given a time-span it decides whether any interference is taking place at the extrema of the time-span, and if not it decides whether any clash could occur within the time-span. If a clash cannot occur the algorithm can finish considering this time-span, and otherwise it divides the time-span up into two new spans and solves the problem within these recursively.
A quantity of the form
\[ \text{distance} / \text{speed} \]
is again used in a test to decide whether a clash might occur, but
now the distance considered is the sum of the minimum distances
between the objects at both ends of the time-span. This is because a
clash can only occur if the objects have time to approach each other
from their initial positions and further have time to retreat to
their final positions. The calculation of the maximum speed is as
for ClashAT(). A pidgin-Algol version of this algorithm is shown
below.

**Boolean procedure ClashDC(obj1, obj2, t0, t2)**

1. \( s \leftarrow \text{maxspeed(obj1, obj2, t0, t2)}; \)
2. \( d_0 \leftarrow \text{mindist(obj1, obj2, t0)}; \)
3. \( d_2 \leftarrow \text{mindist(obj1, obj2, t2)}; \)
4. \( \text{if } d_0 \leq 0 \text{ or } d_2 \leq 0 \text{ */ clash occurs */ return true; } \)
5. \( \text{elseif } s \cdot (t_2 - t_0) < (d_0 + d_2) \text{ */ clash cannot occur */ return false; } \)
6. \( \text{else begin } \)
7. \( t_1 \leftarrow \frac{(t_0 + t_2)}{2}; \)
8. \( \text{return( ClashDC(obj1, obj2, t0, t_1) or ClashDC(obj1, obj2, t_1, t2)); } \)
9. \( \text{end } \)
10. \( \text{endproc } \)

ClashDC() is a better behaved algorithm than ClashAT() in the
sense that the algorithm will terminate if the objects inter-
penetrate. However ClashDC() may recurse indefinitely if the objects
just touch, and any implementation of ClashDC() will be susceptible
to numerical problems when dealing with situations where the objects
either nearly clash or clash for a short period of time. Some com-
parisons of ClashAT() and ClashDC() will be made in section 9.5.
9.3 The Basic Minimum Distance Algorithm

This section describes the basic version of the minimum distance algorithm that I have implemented in robmod, and in the next section I will describe some extensions to the algorithm that speed it up. Formally the minimum distance between two sets of points A and B is given by

\[ \text{md}(A,B) = \text{glb} \{ d(x,y) \mid x \in A \text{ and } y \in B \} \]

where glb denotes the greatest lower bound of a set of numbers. This definition is precise but it does not tell us how to compute this quantity for a pair of modelled objects. I shall state without proof that if the sets of points are closed and bounded sets then there exists at least one pair of boundary points \((x,y)\) with \(x \in A\) and \(y \in B\) such that \(d(x,y)\) is \(\text{md}(A,B)\), and so it is certainly true that for \(r\)-sets we have only to consider points lying in the surfaces of the sets when looking for the minimum distance.

The algorithm that I will discuss uses a segmentation of the surfaces of the objects. Imagine that we have lists of the vertices, edges and faces that bound each of the sets A and B. Then if the sets are not interpenetrating there exists at least one pair of points \((x,y)\) with \(x \in A\), \(y \in B\), and \(d(x,y)\) equal to \(\text{md}(A,B)\). Then each of \(x\) and \(y\) must lie on either a vertex, an edge, or a face. Thus the line from \(x\) to \(y\) connects a pair of entities whose types must be found in the following list:

1. Vertex-Vertex
2. Vertex-Edge
3. Vertex-Face
4. Edge-Edge
5. Edge-Face
6. Face-Face

So a sufficient algorithm to compute \(\text{md}()\) would just find the minimum distance between all the entities of these types. In practice this

\[ \text{Proof is by contradiction - if either point is not a boundary point then the straight line between these points must intersect a boundary.} \]
means calculating for each candidate pair of entities a set of possible minimum distances. Each element of the set would correspond to a pair of points; both points would be derived from each entity but would not necessarily be a part of the entity. This is because the geometry of the entity is normally given as a set of equations which represents a point, line, or surface, and then the boundary of the entity itself is given by the topological data associated with the entity. As an example consider figure 9.4 which shows a circular arc and a straight line; then the shortest distance between the circle and the line is along the dotted line shown, but the left end point of this line does not exist on the circular arc. The true minimum distance between the arc and the line could however be detected by a vertex to edge test.

In the general case for each pair of entities that we consider we have to find the set of local minima for the distance function and then check to see if the end-points that correspond to each local minima exist. If none exist, then the minimum distance will be picked up by a test between a different pair of entities. An example of an edge-edge test that needs to consider two local minima is shown in figure 9.5; most modelling systems that are in current use deal with entities that would only produce one global minimum for each pair of entities tested.
robmod uses a data-structure called a **body** when computing the minimum distance function. A body contains a CSG description of a shape plus a wireframe for the shape. (It also contains other information that will not for now concern us.) The wireframe is just an unordered list of all the edges in the surface of the shape, and it is created by a modified version of the robmod drawing algorithm. The lists of vertices and edges that are required for the minimum distance computation are taken from this wireframe; the list of surfaces is generated by listing all of the surfaces in all of the primitives in the CSG description. Then to check whether a point exists on a vertex entity one has a trivial test; to check whether a point exists on an edge entity I check to see if it lies between the two vertices; but to check whether a point exists on a face I use the ray-casting algorithm that was described in Chapter 4. This means that robmod does not have to create a fully linked boundary model.

In fact a further simplification is possible in the current version of robmod as this version deals with only a polygonal approximation to bodies. Thus all the edges in the wireframe are straight and can be represented by just the coordinates of two points, and further all the surfaces are planar. This in turn implies that cases (5) and (6) in my list of entity type pairs are redundant, as any minimum distance that exists as such a pair must be duplicated by one of the other cases. This leaves just four cases to consider, and they are
considered in the order vertex-vertex, vertex-edge, edge-edge, and vertex-face. I consider the entity pairs in this order and keep a running tally of the smallest distance found so far. Then any candidate distance that is found that is larger than the current minimum can be ignored without having to check the end-points for existence. This is particularly useful for the face-vertex case as the test for a point to exist on a face is non-trivial. The arithmetic required to calculate the various candidate distances between these four types of entity pairs is described in Appendix F.

9.4 Improving the Minimum Distance Algorithm

The problem with the basic minimum distance algorithm is that it does an exhaustive search of the body data-structures. In Chapters 4 and 7 I have described algorithms that use the notion of localisation to quickly concentrate on a small subset of a database; such algorithms are significantly faster than the standard algorithms when applied to models of real objects. The application of the notion of localisation was possible because the problems that were being solved (e.g., WIREFRAME and NOD) were capable of decomposition into smaller problems so that each sub-problem was concerned with just a small region of space. However this is not possible with minimum distance because the property of a point being at the end of a line of minimum distance is not decidable on a local basis. We cannot just use the divide-and-conquer paradigm to concentrate the attention of the algorithm into small regions of space. Instead I have tried a different approach to speed things up.

Consider the view of the Puma robot arm that was presented in figure 9.1. The arm is modelled in robmod as an assembly, which is a collection of bodies with each body having a rigid-body transformation applied to it. Each link in the robot is modelled by one body, but for many cases this bound is pessimistic. A similar situation arises if a boundary model is used as the basis for the test, as then we generally have many edges around a face.
and the robmod transformation chains that denote the transformations that apply to each body in the assembly are written down as functions in the six joint angles of the robot. From a study of figure 9.1 it it "obvious" that the minimum distance between the block and the robot arm will almost certainly occur to a point in the hand of the robot, as the robot hand is that part of the robot that is "closest" to the block. I have added this type of reasoning to the robmod implementation of the minimum distance algorithm by considering, in a rough and ready sense, the distances between the various bodies in two assemblies. To facilitate this a piece of extra information was planted into the body data-structure, namely four parameters that describe a circumscribing sphere around the body. Then the improved algorithm orders pairs of bodies according to a notion of closeness. The circumscribing spheres for the Puma model are shown in figure 9.6.

![Diagram of Puma robot with circumscribing spheres around the bodies.](image)

**Figure 9.6**

†. These spheres are created in a simple-minded fashion by placing a box around the body, and circumscribing the box.
There is a problem in deciding which measure of "closeness" to use when ordering the body pairs. The present implementation calculates three numbers for each body pair. One number is the shortest distance between the circumspheres, which is just the distance between the centre of the spheres minus the two radii of the spheres. This number is a lower bound on the value of the minimum distance between these two bodies. Another number is the greatest distance between the two circumspheres, and this number is an upper bound on the minimum distance between the two assemblies. The third number is the estimated value for the minimum distance between the bodies, and it is chosen by experiment to have a value in the range given by the other two numbers. The estimated value is used to order the list of body pairs, and the current robmod implementation calculates it as the value of the expression

$$D - \frac{1}{2}(r_1 + r_2)$$

where \(D\) is the distance between the centres of the spheres, and \(r_1\) and \(r_2\) are their radii.

The reason for calculating the shortest minimum distance and the largest minimum distance consistent with the circumspheres is that the final version of the minimum distance algorithm can use these to quickly ignore many body-pair instances. The complete algorithm is shown below.

```plaintext
real procedure mindist(assemblyA, assemblyB, t)
    if assemblyA and assemblyB interfere at time t then
        return 0;
    /* the interference test is as described in Chapter 7 - it is a standard function in robmod */
    cur_min <- \infty;
    bodypairlist <- \{\};
    foreach bodyA in assemblyA at time t do
        foreach bodyB in assemblyB at time t do
            begin
                lbd <- smallest possible minimum distance between
```
bodyA and bodyB by sphere test;
ubd = largest possible minimum distance ... ;
ed = estimated minimum distance ... ;

if lbd<cur_min then
begin
  place the pair \{bodyA, bodyB\} in bodypairlist,
  ordered by the estimated distances;
end
if ubd<cur_min then
begin
  cur_min = ubd;
  remove any body pair from bodypairlist which
corresponds to a pair of bodies which are
certainly more than cur_min apart;
end
end

/* by this point we have created the complete
body-pair list */

foreach case of body-entity type pairs do
  foreach bodypair in bodypairlist do
    if smallest possible distance between the
    bodies in bodypair < cur_min then
    begin
      update cur_min by considering candidates of
      the current body-entity type pair;
    end

  return cur_min;
endproc

This algorithm is an improvement on the naive algorithm, but the
use of spheres as approximations for the bodies has not proved to be
ideal for the bodies considered in my experiments. Spheres are not
very good approximations to these bodies and when the assemblies are
not very close together more body pairs are constructed than I would like. As an example of this behaviour the situation in figure 9.6 results in all the bodies of the robot arm from the upper arm downwards being considered. However, when the objects are close together then fewer body pairs are created and the time to calculate the minimum distance decreases appreciably. I suspect that a solution to this problem would be to use a better fitting approximation to the bodies. For the Puma model rectangular blocks aligned with each particular body would fit well, and although this would complicate the calculation of the approximate distances \( mb, ubd, \text{ and } ed \in \text{min-dist}() \) this would not, in fact, degrade the computational complexity of the algorithm. This modification has not been implemented in robmod.

A further way of improving a certain type of minimum distance calculation has been considered. When running ClashAT() it is found that the points on the assemblies between which the minimum distance occurs do not tend to move very much between successive iterations of the algorithm. This leads to an idea of a minimum distance function with a memory, which could use information calculated during its last call to direct its search for a new minimum distance. At the very least such a function could use the new distance apart of the entities which yielded the minimum distance during the last iteration as a first guess of the new minimum distance. Further modifications along these lines have also been considered that should work for convex or "nearly convex" objects. Consider figure 9.7. Two corners are shown which correspond to the nearest points between the regions A and B. Then the hatched region between A and B intersects neither A nor B, and so \( md(A,B) \) must be at least the value of the width of this "barrier" region. So whilst the minimum distance between A and B does occur between these two corners the construction of such a barrier region could be used as a quick test to see whether an exhaustive minimum distance calculation can be avoided.
Figure 9.7

9.5 Comparisons

Both ClashAT() and ClashDC() have been implemented in the ROBMOD language so that their performance could be examined, and for completeness a couple of simpler algorithms have also been implemented. Clashlin() is the simplest multiple interference detection algorithm; given a time interval, \([t_s, t_f]\), and a time step, \(\Delta t\), it checks for interference at times of the form \(t + n\Delta t\), for non-negative integers \(n\) such that \(t + n\Delta t \leq t_f\). Clashran() is similar, but chooses times of the form \((1-\lambda)t_s + \lambda t_f\) for \(\lambda = 0, 1, 1/2, 1/4, 3/4, 1/8, 3/8, 5/8, \ldots\) until it has achieved the required time granularity.

The Puma and workstation model (cf. fig 9.1) was used as the basis of a small set of benchmark tests for these algorithms, with four similar motions being chosen for the robot arm to run through. The motions were all created by running the waist, shoulder and elbow joints of the robot at constant angular velocities, and the motions only differ in the constant of proportionality chosen for the shoulder joint velocity. Figure 9.8 shows two views of the workstation, each view consisting of six snapshots of the robot during just one of the motions. Each view shows the lower-arm assembly of the
robot at times 0, 1, 2, 3 and 4, and also shows the entire robot and workstation at time 5. Figures 9.9a–d show snapshots of the lower-arm assembly at time 1.2 for each of the four motions, and I will refer to the four motions as motions (a), (b), (c) and (d). Motions (a) and (b) do result in a clash between the robot and the block, with motion (b) only just resulting in the two assemblies interpenetrating. Motions (c) and (d) do not result in clashes, with motion (c) giving a near-miss. All of these motions are considered over the time period \([0,5]\), and the number of iterations for each algorithm and motion are tabulated below. At in Clashlin() and Clashran() was chosen (arbitrarily) at 0.1, or 2% of the total time span, and ClashAT() was only expected to get within a small distance of a clash (approximately 0.1% of the total dimension of the robot). Note that even Clashlin() and Clashran() did happen to discover that
there was a clash in motions (a) and (b).

<table>
<thead>
<tr>
<th>Motion</th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clashlin()</td>
<td>12</td>
<td>12</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>Clashran()</td>
<td>3</td>
<td>41</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>ClashAT()</td>
<td>6</td>
<td>7</td>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td>ClashDC()</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>7</td>
</tr>
</tbody>
</table>

These numbers are not particularly meaningful in themselves, but they do illustrate a few points. Firstly, the number of iterations does peak for all the algorithms for the motions (b) and (c) as it is for these motions that it is difficult to decide whether or not a clash occurs. Secondly, all the algorithms do do better on the motions that result in a clash, namely (a) and (b), compared with motions (c) and (d). (This effect is particularly noticeable for Clashlin() and ClashAT() because the clashes happen to occur early in the motion.) Thirdly, "luck" can sometimes play a part; the number 3 for Clashran() and ClashDC() with motion (a) occurs because for this motion there happened to be a secondary clash occurring at time 2.5.
10. Four Dimensional Modelling and Sweeping

This chapter is concerned with the other two methods for implementing clash detection that I have considered, namely by four-dimensional interference detection and by sweeping. In the former the clash detection problem is transformed into a problem of looking for interferences between four-dimensional entities. The extrusion of a four-dimensional set from a three-dimensional set and a location function was discussed in Chapter 2, and the use of such worms for clash detection was mentioned in section 3.7. Here I am concerned more with the practicalities of the method, and to facilitate an understanding of these the method has been implemented in robmod for a certain class of allowable motions, namely, piecewise constant-velocity motions. The implementation is described in sections 10.1-10.4, and section 10.5 considers some extensions to this implementation.

No implementation of a method involving sweeping exists in robmod, but sections 10.6-10.8 consider this method in the light of other research. The chapter concludes by highlighting some of the similarities between these two methods.

10.1 Describing Motions

For simplicity the current implementation of four-dimensional interference detection deals only with piecewise constant-velocity motions; that is, at all but a finite number of times the motion is a constant velocity, and these motions can be described in the ROBMOD language by an entity of type motion. A motion can be written as a motion chain, which is a list of primitive motions and the times for which they are applicable. Each primitive motion is described by a velocity, which is written as a triple after the keyword vel, and the

†. As it is always possible to distort the time scale this is not quite as restrictive as it might at first appear.
word **rest** is available as a synonym for **vel 0 0 0**. Examples of valid ROBMOD motion chains include

```
vel 1 2 3
and rest until 0 vel 1 2 3 until 10 rest
```

The first example denotes a motion which is just a constant velocity \((1,2,3)\) for all time, and the second denotes a motion which is a constant velocity in the time interval \([0,10]\), and no movement for all other times.

A shape in motion is described in ROBMOD as an entity of type **worm**, which is a pair with one component an **assembly** and the other a **motion**. A worm entity may be written as

```
assembly expression via motion expression
```

Here the motion expression is akin to the location functions discussed in Chapter 2, but there is a subtle difference in that the motion is in fact the derivative of the location function w.r.t. time. To fix a particular location function to a worm a default is assumed, namely that the location function returns the identity transformation when time is set to zero.

As an aid to visualising worms the function **slice** is available in ROBMOD, which takes a worm and a time and returns an assembly that describes the assembly component of the worm at that time. The assemblies returned by **slice** are generally used to draw a picture of a situation at a particular time.

### 10.2 Creating Four-Dimensional Models

The worms in ROBMOD describe the entire history of their objects, and so contain all the information required by the clash detection routine. There is a clash detection function in ROBMOD that acts on worms; it is called by the command
\texttt{clash \ t_s, t_f, wormA, wormB}

where $[t_s, t_f]$ is the time period over which the search for a clash should be conducted.\textsuperscript{†} The worms arrive at the clash detection function as a shape/motion pair (the extra information in the assembly data-structure is not required here), and these pairs are extruded to form two four-dimensional CSG trees. The process that creates these four-dimensional trees will be discussed below. These trees are spliced into a new tree with an intersection node at its head, and then further intersected with four-dimensional half-spaces that enforce the time limits; this is shown diagrammatically below.

![Diagram of four-dimensional CSG trees](https://via.placeholder.com/150)

The tree that describes a worm is formed by taking the union of $n$ trees, where $n$ is the number of separate primitive motions in the motion chain component of the worm. For example, the worm that is described in ROBMOD as

\texttt{assembly via rest until 0 vel 1 2 3 until 10 rest}

will produce a CSG tree that is the union of three smaller trees. In turn, each of the smaller trees will be the intersection of a subtree describing a time-span and a subtree describing a shape with the required motion. As an example, the ROBMOD expression above yields a tree of the form

\[ \texttt{A = A \lor A \lor t_s \lor t_f} \]

\textsuperscript{†}. For most calls to clash the time bound is unnecessary as a suitable bound could be inferred from the motion components of the worms; for example, clash detection need not be performed for any time for which both of the motions are \texttt{rest}.
with the subtrees A, B, and C describing the shape extruded under different location functions. To form these subtrees the Distribution Theorem is used (Theorem 2.4). The CSG tree describing each (three-dimensional) shape is copied, and each leaf of this tree contains the geometry of a surface. In the present version of robmod all cylinders and cones are replaced by polyprisms at this level, and so the only sort of (three-dimensional) surface that this implementation has to deal with is a planar surface, which corresponds to a half-space of the form

\[ p \cdot x + d \geq 0. \]

where \( \cdot \) denotes the scalar product operation on vectors. The location function for the worm is a translation of the form

\[ c + vt \]

where \( v \) is the velocity at this part of the motion and \( c \) is calculable from the constraint that the composite motion must yield an identity transformation at time zero, and so the extrusion of this space is

\[ p \cdot (x - c - vt) + d \geq 0 \]

or \[ p \cdot x - (p \cdot v)t + (d - p \cdot c) \geq 0 \]
which is a (four-dimensional) planar half-space (see section 2.3 for a discussion of this transformation). Then to form the four-dimensional CSG trees this transformation is applied to all the face-geometry nodes in the trees.

10.3 Four-Dimensional Null Object Detection

The tree which is created by the mechanism described in the previous section is fed to a four-dimensional NOD routine. This routine is currently a four-dimensional implementation of Bath_NOD(), and the robmod routines that implement Bath_NOD() in three and four dimensions have much in common. Bath_NOD() requires a box in which it must test for an object being null. The temporal extent of this box is just \([t_s, t_f]\), and the spatial extent is calculated by boxing each object separately at time \(t_s\), time \(t_f\), and at all the times at which the motion of that object changes in \([t_s, t_f]\). This gives a temporal and spatial bound on each worm, and these two bounds are intersected to form a final bound for Bath_NOD(). A more intelligent bounding scheme could be implemented using S-bounds, as will be mentioned in section 10.5.1.

The only real difficulty with the implementation comes in the handling of the base case, that is, in baseNOD() of section 7.1. In the three-dimensional case baseNOD() proceeds by trying to discover points that can be proved to be either inside or on the "null" object, and this is done by taking pairs of surfaces from the CSG tree to form candidate lines, and then intersecting each line with all the other surfaces to form candidate edges, and finally testing the mid-points of these edges by point membership classification (PMC). After some deliberation a similar strategy was used in four-dimensions; triples of hypersurfaces were intersected to form candidate lines, then these were intersected by other hypersurfaces to form candidate edges, and then the mid-points of these edges

\[ \dagger \] A slightly more complicated boxing procedure may be required in an implementation in which the primitive motions are not constant velocities.
considered. The problem is to implement PMC in four dimensions. In fact for most of the calls to Bath_NOD() I have observed that the PMC required is trivial, as the simplified CSG tree that describes the object in the neighbourhood of the point turns out to be $T$ or $\perp$; further I have developed closed-form solutions to this particular problem when there are three or fewer distinct surfaces passing through the point. This apparatus has proved sufficient to solve all of the calls to clash so far given to robmod; the general case (of four or more distinct surfaces at a mid-point of an edge) has also been solved in theory, but not implemented yet in robmod.† A discussion of the solution to PMC in this domain is to be found in Appendix G. Note that although the solutions have only been implemented for planar surfaces, they could in fact be modified for other surfaces in a systematic manner, although such a modification would increase the amount of code written considerably.

A further modification that can be made to this algorithm would enable one to obtain the time of the first clash. The modification is akin to the difference in three dimensions between drawing an object and performing NOD on the object, as we have to find all of the edges that bound the (four-dimensional) object that "represents" the clash (the "clash volume"), and thus find the point that is earliest in time. This approach could be speeded up slightly by careful ordering of the division steps in Bath_NOD(). If the hypersurfaces involved in this operation are not planar then the situation may be complicated slightly, as then the point with minimal time coordinate need not lie on a vertex.

10.4 Examples

A cartesian robot was invented as a suitable test for the algorithm. The basic robot is shown in figure 10.1; it consists of an arm which can be moved up and down in a trolley, and the trolley can move in a horizontal plane above the table upon a gantry. Figure 10.1. The solution relies on a duality between PMC on an edge in four-dimensions, and PMC at a vertex in three-dimensions. It is messy.
10.2 shows the workstation which contains three loaded pallets running on tracks, and a couple of feeders fixed at one end of the table; the gantry and trolley mechanism have been ignored for clar-
lity. The test scenario was of the robot arm being required to make a point-to-point motion between its rest position (as in figure 10.1) and its final position above a feeder (as in figure 10.2). The arm is then in danger of clashing with the loaded pallet in the middle of the table. robmod allows the motions of the arm and the pallet to be altered interactively and allows pictures of the situation to be drawn at any simulated time, and so the user is able to form his own opinion as to whether a clash occurs, or he can ask robmod itself. Various motions were given to these two assemblies and the clash function solved all the problems that it was given correctly. Indeed the function has often proved itself to be more capable of testing for a clash than the author. The speed of the function is, subjectively, quite encouraging, for it appears to give the same sort of performance that I have come to expect from the three-dimensional version of Bath_NOD().

Other configurations have been run on the system without problem, and note in particular the one shown in figure 10.3. It shows a rectangular peg passing through a rectangular hole, and it is an example for which all my multiple interference detection algorithms are ineffective.

Figure 10.3
10.5 Extensions

10.5.1 Improved NOD

The current implementation of four-dimensional NOD is adequate, but relatively simple as all of the methods described in Chapter 7 are applicable to four-dimensions. It is a consequence of the CSG-tree creation algorithm of section 10.2 that a motion with a large number of motion components will give rise to a proportionately large CSG tree; the effect of these large trees could be minimised by creating bounds around the primitives of the tree which extend in time only for as long as their motion components are applicable. Then the S-bound schemes of Chapter 7 should give the same sorts of advantages as they have been shown to do in three dimensions. However, care would have to be taken when creating the bounds that will be used. As an example, if we have a unit cube \([0,1] \times [0,1] \times [0,1]\) with a velocity \((1,1,1)\) and consider it in the time interval \([0,9]\), then a bounding box for the corresponding worm will be the space \([0,10] \times [0,10] \times [0,10] \times [0,9]\); but the worm only occupies 0.1\% of the hypervolume of this box. This is the analogue of the problem in three-dimensions of trying to ensure that the bounds that are chosen fit most of the objects snugly. A better approach here might be to cover the worm by a set of boxes, say \([0,2] \times [0,2] \times [0,2] \times [0,1]\), \([1,3] \times [1,3] \times [1,3] \times [1,2]\), ...}.

10.5.2 Wider Classes of Motions

The motions allowed in the robmod implementation of this method were chosen so that the resulting half-spaces were relatively easy to describe and manipulate. For clash detection involving objects which have motions that are mainly translational in nature one might approximate those motions by piecewise constant-velocity motions. Another approach would be to permit a larger range of surface types
in the models of the worms, and work that is now being done on allowing "sculptured" surfaces in modelling systems may soon make this practical for some motion classes (e.g. [Franklin & Barr 81]). However motions that are largely rotational in nature give rise to hypersurfaces that will probably prohibit their direct use in this method for the immediate future. (There is a very interesting exception to this statement though. A cylindrical peg rotating about its axis of symmetry is preserved under extrusion.)

10.5.3 Better Motion Descriptions

An extension that could be added to an implementation without much difficulty would be to allow a wider range of ways of describing motions. An obvious example would be a way of defining a motion from one location to another at a constant velocity. Another example would be to allow the motion of the cartesian robot to be denoted by functions that describe where the gripper is. Such extensions have immediate practical value, but were considered to be outside the scope of this research.

10.6 Sweeping Methods

The principle behind this third method for performing clash detection is that of characterising the entire space that is ever occupied by a moving object. If we have a set $S$ which has a motion given by a location function $L$ then we can define an operator $Sw$ by

$$Sw(S, L) = \{ x \mid x \in L(t)(S) \text{ for some } t \text{ in } \text{TIMES} \}$$

and we say that the (three-dimensional) set $Sw(S, L)$ is the sweep of $S$ (under $L$). Then if two objects clash their sweeps must intersect. We can show that $Sw$ preserves regularity by the theorem below; the proof is given in Appendix A.
Theorem 10.1

If $S$ is a regular set and $L$ is a continuous location function then $Sw(S, L)$ is a regular set.

The simplest way to utilise sweeps for performing clash detection is to model the sweeps of the objects, and then to look for intersections between them. (As was explained in section 3.8 this may be complicated slightly if the objects are both moving as we then have to consider the relative motions of the objects.) I will call this method explicit sweeping, and it is discussed in the next section. The other variant that I will consider is implicit sweeping, in which no single three-dimensional model of an object is created. This latter method will be seen to have advantages over the former in many circumstances, and implicit sweeping is discussed in section 10.8.

10.7 Explicit Sweeping

The problem with explicitly creating sweeps of objects is that no algorithms have been found that will do this for anything other than a small class of objects and motions. Four-dimensional interference detection was practical for CSG-based representations because of the use of the Distribution Theorem, but no similar theorem holds for $Sw$. For example figure 10.4 shows two planar sets, $A$ and $B$, with $A\cap B = \emptyset$. But if $L$ is a location function that translates $A$ and $B$ along the vector shown then
\[ Sw(A, L) \land Sw(B, L) = \emptyset. \]

A similar counter-example can also be found for the difference operator (figure 10.5). In a boundary modelling context the situation is just as bad, as much of the topological information stored in a boundary model will be useless in building up a boundary model of the swept volume.

However, \( Sw \) is distributive over union, and so if the objects can be considered as the union of a number of parts and these parts can be swept then the sweep of the entire volume can be calculated. An example of this approach is to be found in the LARS system [Balila & de Pennington 83]. LARS is a system built over the NONAME geometric modelling system; NONAME is a CSG-based, industrial-grade system which has been developed by the Geometric Modelling Project at the University of Leeds. LARS is described by Balila as an "off-line robot system", and to perform clash detection he can model robots approximately as the union of a number of spheres and sweep these along a linear trajectory or along a rotational trajectory. In the former case the swept volume of a single sphere is modelled as the union of a cylinder and two spheres, and in the latter case the swept volume is modelled by a segment of a torus and two spheres. All of these are available as primitive shapes in NONAME.
10.8 Implicit Sweeping

The creation of an explicit sweep provides a neat solution to the clash detection problem when it is practical, but unfortunately this is not always the case. By implicit sweeping I mean a method that relies on the swept-volume paradigm but does not create a single model of the sweep. An example should help to clarify this statement. In [Boyse 79] a system is described that checks for clashes between polygonal objects moving along a linear path or rotating about a fixed axis (as in LARS). The representation scheme was a form of boundary model, and instead of creating a single new model for the swept volume Boyse took each entity from the boundary model and swept them individually. In effect he was looking for clashes between the boundaries of the objects, and using the fact that the union of the sweeps of all the boundary entities is equal to the sweep of their union.

Boyse discovered that he could simplify matters in the case of planar objects, as then he had only to look for the sweep of a vertex intersecting a fixed face or the sweep of an edge intersecting a fixed edge. In turn this implied that for the motion classes that he considered he had only to test for a straight edge or circular arc intersecting a polygon, or for a straight edge intersecting a hyperboloid of revolution. As a further development of the system he could ask for all the possible clashes to be found, and then just report the earliest. This is something that it is difficult to do in a system that does clash detection by explicit sweeping as there is then little correspondence between parts of the object and parts of its sweep.

The implicit sweeping method does have its problems though. Because of his choice of planar surfaces Boyse only had to ever test for intersection between a two-dimensional entity and a one-dimensional entity. If one were to add just cylindrical surfaces to this scheme it would be necessary to consider the possible intersection of pairs of three-dimensional surfaces. It might also make it
harder to extract the time of a clash due to there no longer being a simple mapping between points in the swept volumes and the time scale. So allowing a larger range of surface types would greatly increase the difficulty of implementing this method. In fact it may be noted that the types of surfaces that would be created in the sweeps are similar to those that would have to be described if such surfaces were permitted in a system that performed clash detection by creating four-dimensional models.

10.9 Summary

The implicit sweeping method follows the computational metaphor

for each entity pair
process the entities
perform an intersection test

which is rather similar to the metaphor

process the entities
for each pair of processed entities
perform an intersection test

For implicit sweeping the processing in the first metaphor consists of sweeping the entities. The second metaphor is followed by four-dimensional interference detection, where the processing consists of extruding the entities. These processes can be performed in different orders because the temporal information in the object/motion pair is not lost during the extrusion process, whereas pure sweeping suppresses this information. In fact sweeping can be considered to be the projection of the extrusion onto the Oxyz hyperplane.† The most noticeable difference between the algorithms for clash detection which are based on four-dimensional interference detection and on

† A common use of a projection operation occurs when a three-dimensional object is projected onto a plane so creating a two-dimensional image of the object. Projection can thus be seen as a form of data-compression
implicit sweeping is that the latter has available information about the vertices, edges and faces in the original object and uses these as foci for the attention of the algorithm. Conversely, the former had less information to worry about but is able to use the methods I have described for implementing NOD, namely the Bath group's divide-and-conquer method and my S-bound method. Doubtlessly there is as much room for disagreement here as to which method is "best" as there has been in the question of whether CSG-based or boundary-based geometric modelling systems are better for drawing.
11. Clash Detection Systems

In the last two chapters I have been taking some of the ideas that were presented earlier in this thesis and using them as parts of algorithms for performing clash detection. The purpose of this chapter is to put forward some ideas for further systems that could be implemented and that could be used to perform clash detection in a CAD/CAM environment. Such systems can range from a human operating a simple CAD system through to an advanced manufacturing planning system. The next four sections will review my three techniques for performing clash detection and then outline how a composite clash detection algorithm that combines two of these methods might work. In the final section I will list some possible configurations for clash detection systems.

11.1 Multiple Interference Detection

Multiple interference detection is a simple technique, but it can be the basis of algorithms that range from the very simple to the very complex. I have looked at two classes of multiple interference detection algorithms, namely unguided algorithms (Clashlin() and Clashran()) and guided algorithms (ClashAT() and ClashDC()). All of these can be made to show a human user the situation at the current simulation time for each iteration of the algorithm (say by drawing a wireframe picture), and thus all of these can, in some sense, account to the user for what they are doing. This question of accountability is an important one; a user who has no faith in an algorithm will cease to use it. In terms of this accountability I have found it easier to follow what Clashlin() and ClashAT() are doing than to follow the actions of Clashran() and ClashDC(), and so it seems as though one of the former pair might be the better algorithm to implement in an interactive system. Clashran() and ClashDC() do have theoretical advantages, namely that Clashran() is expected to find a clash faster than Clashlin() and that ClashDC() may find clashes that
ClashAT() cannot, and so Clashran() and ClashDC() may be the more natural choice in a non-interactive environment. Note that all of these algorithms can be adjusted to hunt for the time of first contact.

Multiple interference detection algorithms do not need to know a motion precisely, and for testing a motion that should be free of any near-clashes the models of the objects that the algorithm uses could be enlarged slightly to ensure that the algorithm terminates without causing it to report too many false clashes. The guided algorithms have the advantage of being able to never miss a clash, and they also tend to not need so many iterations as their unguided counterparts. It is difficult to say much about the expected complexity of these algorithms; however, it is true that the present iteration time for my guided algorithms are significantly larger than for their unguided counterparts as the time to execute my implementation of mindist() dominates the former. Possibly this situation could be improved by using a more coarse measure of "distance" in the guided algorithms. An example of this approach would be a domain consisting of objects resting on a plane, and in this domain a better distance function might be the distance between the projections of the objects onto the plane. Such improvements are, of course, highly dependent on the domain of interest. Conversely, in some situations it might pay to have a more intelligent algorithm that could take the directions of the motions into account. A problem with my guided algorithms has been their tendency to get bogged down when there are small separations between objects, and to stop altogether on contacting motions. There is also probably some room for improvement by using approximations to objects to help speed up the cycle times; an example of this approach is given in section 11.4.

11.2 Four-dimensional Interference Detection

The idea of modelling an object and its motion as a single entity was first noted in chapter 2 of this thesis where it was attractive because of its mathematical properties. Unlike multiple interference
detection the use of four-dimensional interference detection as a basis for clash detection implies that the motions have to be describable to such a system both at the input level and at the internal level of the four-dimensional model (it is not enough to sample the motions). This requirement may lead to the sort of restriction that is imposed on the motion entity in robmod, namely that the motions be piecewise constant-velocity. A useful extension would be for such a system to itself linearise an input motion and so allow four-dimensional interference detection to be used as an approximate method for a wider class of motions.

Four-dimensional interference detection does have an advantage in that it can be implemented using the techniques of Chapters 4 and 7. I have not attempted a close statistical analysis of this algorithm along the lines of Chapter 8, but note that if the constant-density assumption of that chapter holds for a set of three-dimensional objects then the expected density of the extrusions of these objects will be constant, and so I would conjecture an expected complexity for (four-dimensional) Bath_NOD() of $O(n \log n)$, where $n$ is the product of the complexity of the object and the number of motion components. However without running experiments on "real" clash detection situations it is difficult to be sure how well the S-bound technique will preprocess the CSG descriptions of the clash "object". The initial impression of the four-dimensional implementation of Bath_NOD() has been encouraging, as the algorithm does seem to work hard only on the truly difficult situations; that is, when the worms just intersect or are close together.

One thing that it is not easy to do with a four-dimensional interference detection scheme is to get it to produce "reassuring" pictures for a human user, and for similar reasons it can be difficult to debug programs that implement this technique. However this effect is offset by the fact that there is no need to worry about the handling of special cases, and in particular sliding contacts are handled well by the system. Faces with symmetries that coincide with an applied motion are preserved under extrusion, and I believe that the exploitation of these symmetries in such a mechanism might be an
interesting extension to this work.

11.3 Sweeping

The idea behind the use of the sweep of an object is that we are characterising all of the volume ever occupied by the object. If we never have to check for any clashes between a pair of objects that are both moving then we can use a single coordinate system for creating the sweeps and we only have to create one sweep for each object (which may be the same as the object itself). Otherwise we may have to create a number of different sweeps for an object.

No general algorithms are known for creating sweeps, but if the sweeps can be generated then explicit sweeping is a neat way of performing clash detection as it is easy to show what the algorithm is doing. Implicit sweeping, however, should have a larger performance envelope, as it is basically performing the same function as an explicit sweep but without having to merge its intermediate results into a coherent model. It is also easier to extract the time of first contact with an implicit sweeping method than with an explicit method.

The representations required to implement a sweeping method are similar to those in four-dimensional interference detection, except that the time coordinate is normally suppressed by a projection operation. Thus a boundary representation of a sweep may amount to less data than a boundary representation of an extrusion, but the former may often be more indigestible as a result. Sweeping may well be a suitable method if the objects can be broken down into a union of easily sweepable parts, even if this can only be done as an approximation.
11.4 A Composite Algorithm

Up to now I have been considering methods for performing clash detection in isolation, but there is no reason why these methods cannot be combined into single systems. In this section I wish to outline how two of the methods that I have discussed might be combined. The general idea is to use multiple interference detection for checking the easy part of a time-span, and then to use four-dimensional interference detection to check the harder parts of the time-span. Also it is intended to use a hierarchy of approximations to the shapes of the objects in order to speed the process up.

Figure 11.1

Consider figure 11.1, which shows a view of the lower part of the Puma arm and a block, together with some circumscribing spheres for the separate bodies. The view is another view of the situation shown in figure 9.9a, chosen so that the approximate distances between the circumspheres is apparent. We can regard the circumspheres as approximations to the bodies, and in the situation shown we can see that the circumspheres surrounding the lower arm link and the wrist do not intersect the circumsphere around the block, whereas the circumspheres around the hand and the fingers do intersect with the circumspheres around the block to various degrees. This suggests that a
suitable composite algorithm might use one of the following three methods for each pair of objects:

1) When the objects are far apart, use an intelligent multiple interference detection algorithm based on the distances between the circumscribing spheres.

2) When the objects are closer, use an intelligent multiple interference detection algorithm based on the distances between the objects themselves.

3) When the objects have nearly linear motions, use four-dimensional interference detection.

I shall expand on the meaning of "nearly linear velocity" at the end of this section. A suitable pidgin-Algol version of this algorithm is shown below as the procedure compclash(), which takes as arguments a time-span and a list of pairs of objects that might clash over this time-span.

```plaintext
Boolean procedure compclash( t₀, t₂, object_pair_list);

if object_pair_list is empty then
    return false;

new_list + [];
foreach {A,B} in object_pair_list do
begin
    if canDo4D( A, B, t₀, t₂) then
    begin
        if clash4D( A, B, t₀, t₂) then
            return true;
        end
    else /* use something like ClashDC() */
    begin /* try a sphere approximation first */
        d₀ = spheredist( A, B, t₀);
```
\[ d_2 = \text{spheredist}(A, B, t_2); \]
\[
\text{if } d_0 < \text{trigger}(A, B, t_0, t_2) \text{ or } \\
\text{d}_2 < \text{trigger}(A, B, t_0, t_2) \text{ then} \]
\[
\begin{align*}
\text{begin} & \quad /* \text{use the real shapes} */ \\
& \quad d_0 \leftarrow \text{mindist}(A, B, t_0); \\
& \quad d_2 \leftarrow \text{mindist}(A, B, t_2); \\
\end{align*}
\]
\[
\text{if } d_0 < 0 \text{ or } d_2 < 0 \text{ then} \]
\[
\begin{align*}
& \quad \text{return true;} \\
& \quad s \leftarrow \text{maxspeed}(A, B, t_0, t_2); \\
& \quad \text{if } (d_0 + d_2) < s(t_2 - t_0) \text{ then} \\
& \quad \begin{align*}
& \quad \text{add } \{A, B\} \text{ to new_list} \\
& \quad /* A \text{ and } B \text{ may clash} */ \\
& \quad \end{align*}
\end{align*}
\]
\[
\text{end} \\
\text{end}
\]
\[
t_1 \leftarrow \frac{t_0 + t_2}{2}; \\
\]
\[
\text{return comoclash}(t_0, t_1, \text{new_list}) \text{ or } \\
\text{comoclash}(t_1, t_2, \text{new_list}); \\
\]
\[
\text{endproc}
\]

For each object pair, the algorithm first decides whether four-dimensional interference detection is applicable (by use of the predicate cando4D()), and if it is this method is used. Otherwise a figure for the distance between the objects is calculated, either from the distances between the sphere approximations if the spheres are not too close together or from the true shapes of the objects if they are close, and this distance is used in a minimum distance/maximum speed test (cf. ClashDC()). If the system is still unsure as to whether the objects clash the object pair is added to a new list of object pairs for consideration in two smaller time spans.
The algorithm as shown is not complete. In a more practical form the algorithm would probably contain some interaction between the control predicates (cando4D() and trigger()), and also the various numbers returned by mindist() would be remembered, rather than being recalculated at the next recursive call to compclash(). However I feel that this algorithm is a useful starting point for future research.

I have suggested that "nearly linear" motions could be handled by a four-dimensional interference detection algorithm, and a suitable measure for "nearly linear" is as follows. One way to represent a transformation is as a translation in space, followed by a rotation about some axis. Thus the motion of an object over a time interval may be written as

\[ \mathbf{x} \rightarrow \mathbf{x} + \mathbf{r} + R(\text{axis}, \theta)(\mathbf{x}) \]

where \( R(\text{axis}, \theta) \) is a rotation about "axis" by \( \theta \) (radians). Then we can say that a motion is nearly linear if the maximum possible error in considering the motion to be the translation

\[ \mathbf{x} \rightarrow \mathbf{x} + \mathbf{r} \]

is small compared with the dimensions of the object. So if the object is associated with a circumsphere of radius \( r \) then the maximum error is \( r\theta \), and as the size of the time-spans gets smaller so must \( \theta \).

11.5 Complete Clash Detection Systems

In previous chapters I have tended to try to remove the need for human intervention in my algorithms; this was prompted by my special interest in process planning systems, but there will be many situations where it will be quite permissible to leave a human in charge of the process. In this section I wish to list some possible systems that involve clash detection. These systems are listed in a rough order of "machine intelligence", but I do not want to claim that this
ordering is necessarily strict.

S1. Visual Checking
This is the simple case of a human operator using a standard CAD system to display objects at different parts of their motions, and from different viewpoints. It is up to the operator to decide whether a clash occurs.

S2. Visual Checking plus Interference Detection
As S1, but the operator can ask the CAD system to perform interference detection at any time, and so such a system must utilise a solid modeller.

S3. Fine Multiple Interference Detection
A simple multiple interference detection routine is run (i.e, Clashlin()) with a fine time step. This system is probably more suited to non-interactive working where the time taken to perform the tests is not critical (EMULA [Meyer 81] is an example of such a system).

S4. Coarse Multiple Interference Detection plus Visual Checking
Here the operator can run a simple multiple interference detection routine with a relatively coarse time step. Furthermore the system gives a quick wireframe picture of the situation at each time step so that the operator can look out for problematic segments of the motion.

S5. Human Controlled Multiple Interference Detection
Rather like S4, but now the operator has a "joystick" that can vary the size of the time step used by the algorithm to both positive and negative values. Thus the operator can get the system to check more cautiously when he is suspicious of the current situation.

S6. Human Controlled Intelligent Multiple Interference Detection
Rather like S5 but now the algorithm used is "intelligent" and the joystick is used to control the program's estimate of the "speed" of the objects (e.g, to set the value returned by maxspeed() in
ClashAT(). This gives the user the ability to chivvy the system when he (the user) can see that caution is unnecessary.

S7. Intelligent Multiple Interference Detection
This system would be something like ClashAT() or ClashDC() but without the interactive features of S6 and including a "proper" implementation of maxspeed(). Like S3 this system is probably more suited to non-interactive use.

S8. Simple (Explicit) Sweeping
Rather like the LARS system ([Balila & de Pennington 83]); an approximation to the shapes of the objects are swept, mainly under human control.

S9. Sweeping plus Multiple Interference Detection
Like S8 but also using a multiple interference detection method to further check any reported clashes.

S10. Complex Sweeping
Such a system would handle "awkward" shapes and motions, and would thus be able to take the relative motions of objects into account. This system would require a relatively complicated internal representation.

S11. Four-Dimensional Interference Detection
At this level this would be a simple implementation of the method and would thus probably be restricted to linear motions, but quite able to deal with cartesian mechanisms.

S12. Improved Four-Dimensional Interference Detection
Like S11, but either capable of representing a larger range of motions directly or capable of linearising such motions.

S13. Multiple Interference Detection plus Four-Dimensional Interference Detection
Rather like S7, but now interactive so that the human user can switch
from one algorithm to the other in the middle of a motion (so he would switch from multiple interference detection to four-dimensional interference detection if the former gets stuck near a possible clash). It would be preferable if the operator could specify which object pairs were to be processed by which algorithm at any time, and an interface via a pointing device (such as a light pen) would seem to be a good solution.

S14. Menu-Driven System
A system that combines a number of different algorithms, complete with a simple programming capability that allows a user to write a clash detection macro. An example of such a macro might be paraphrased as "use ClashAT() initially, and if the distance between the pair of objects falls to less than 2% of its initial value then use four-dimensional interference detection". Then the user would build himself up a library of such macros, and select the one that he feels is most appropriate for a given situation. (The algorithm comp-clash() could be taken as a first attempt at a system with capabilities at this level.)

S15. Intelligent Menu-Driven System
Just like S14, but with the ability to handle assemblies and mechanisms.

S16. Semi-Autonomous System
A system with the same "macros" as S15, but able to choose the macro(s) it will use itself and then to ask for further assistance only if it cannot solve the problem. Such a system would have to be able to recognise what type of clash detection problem it is that it has been given, say by being able to recognise that a problem matches a certain template (e.g., "spherical robot doing a coarse point-to-point motion").

S17. Fully-Autonomous System
The ultimate goal. A system rather like S16, but able to sort itself out if it gets stuck, and able to learn how to create new "macros"
for itself.

This list is by no means exhaustive as other combinations could doubtlessly be tried. The main problem with the systems that appear early on in this list is to engineer a decent human interface. Towards the end of the list the problems become less well-defined, and it is certain that the implementation of S15, S16 and S17 would constitute major research efforts.
12. Conclusions and Further Research

In this thesis I have described some of the problems that have to be faced when adding temporal information to geometric models. In particular the clash detection problem and the domain of robotics have been used as focus for this research, and so the thesis has delved into many areas from these two fields. Nevertheless it is hoped that the work reported will be useful to many, rather than just being of interest to a few specialists.

In this chapter I wish to emphasise the contributions that have been made in this thesis, and to provide some suggestions for future research.

12.1 Contributions

The major contributions of this thesis can be conveniently be grouped into five areas.

12.1.1 Four-Dimensional Modelling

I have extended the idea of a three-dimensional r-set, as used for describing shapes, into the idea of a four-dimensional r-set that describes a shape in motion. The fact that all of the nice properties of three-dimensional r-sets extend to the four-dimensional extrusions has meant that I have been able to use these extrusions with some confidence, and in particular my Distribution Theorem means that we can create CSG models of extrusions (which I called worms) in a straightforward manner. In turn this has enabled the implementation of clash detection by four-dimensional interference detection.
12.1.2 S-bounds

S-bounds are probably the most immediately useful result of this research. They can be seen to be related to the "boxes" and "bubbles" that are used for efficiency in modelling systems, and given a CSG representation S-bounds can be refined by a set of very simple rules based on tree walking. Despite this simplicity they exploit the constructional information that is present in a CSG representation to produce bound sets that are especially useful for intersection problems. In contrast to their ease of use I have found it necessary to establish a rigorous framework to prove that they do, in fact, work; this framework incorporates notions such as "CSG trees" and "tree substitutions".

12.1.3 Null Object Detection and Interference Detection

My algorithms for null object detection and interference detection include some significant extensions to previous work. The basic null object detection algorithm is an adaptation of the drawing algorithm that was developed at the University of Bath, and it has also been extended to work in four dimensions. The redundancy-based pre-processing stage for the final NOD algorithm is based on research that was carried out at the University of Rochester, but I have extended that work with the notion of a covering set of S-bounds. My interference detection routine benefits from these changes by typically being able to quickly dismiss large volumes of space that would have otherwise been considered.

12.1.4 Intelligent Multiple Interference Detection

In this research I have shown how multiple interference detection algorithms can be endowed with a modicum of intelligence by the use of limits on the internal time steps; these limits are based on the
speed of objects and their distance apart. I have also described implementations of both a basic minimum distance algorithm, and also a minimum distance algorithm which makes use of heuristics. These algorithms have been implemented in the robmod system and using the facilities of the ROBMOD language.

12.1.5 Statistical Analysis

The statistical analysis of computational geometry algorithms is still a young area of research. I have taken Tilove's recent analysis of the WIREFRAME problem, and I have extended it in two important respects. Firstly, I have included an analysis of the Bath group's tree simplification routine, which turns out to have a better complexity than Tilove's simplification routine, and secondly I have allowed variable sized primitives in the squares world (using a log-normal distribution for the sizes).

12.2 Topics for Further Research

Finally I will list some possible extensions to the work reported in this thesis.

12.2.1 Describing Motions

The descriptions of motions is paramount to the modelling of solids in motion. Two examples of direct textual descriptions of motions have been employed in ROBMOD, namely as explicit equations containing a variable that stands for time, and as chains of primitive motions. Could we also create useful input schemes using more convenient input methods, such as interactive graphics or natural language? Can we devise such systems that are general, usable and yet capable of creating any other internal representations that may be required?
In chapter 2 I discussed the properties of extrusions; can we construct a useful closure class to describe the r-sets that we normally wish to produce? Such a class would include all of the semi-algebraic sets, but would probably be only a proper subclass of the class of semi-analytic sets.

12.2.2 Four-Dimensional Modelling

Can we represent enough classes of hypersurfaces to make exact modelling of extrusions practical, or should we be prepared to approximate such extrusions? If we choose to model extrusions exactly can we make the necessary extensions to the procedures (such as baseNOD()) that have to deal with the hypersurfaces? Conversely, if we approximate the hypersurfaces, say by linearising the motions, can we create procedures that effectively generate such approximations? And can we exploit the preservation of symmetries under extrusions?

12.2.3 Representations and Basic Algorithms

In this thesis I have concentrated on CSG representations. Could we devise procedures that will rewrite CSG trees into more useful, equivalent trees? Would the use of an explicit boundary representation help, especially in routines such as those to calculate the minimum distance function? And could other representations be used, such as a representation in which a shape is expressed as the union of a number of convex parts, or a representation in which a shape is expressed as an alternating sum of parts ([Woo 82])?

Divide-and-conquer has been seen to be a useful device for solving geometric problems. If we are using a divide-and-conquer algorithm to perform a geometric task (such as Bath_NOD()) can we tune the control structure to improve the overall efficiency of the algorithm? Could we create a self-tuning control structure — that is, a control structure that could learn?
In this thesis some use has been made of approximations to the shapes of objects. Can we lay down guidelines for the use of approximation hierarchies to objects and/or motions?

12.2.4 S-bounds

In this thesis I have shown that the refinement steps that I use for S-bounds are valid. Can we derive estimates for how good a bound set we can create using an algorithm based on S-bounds? Do the sets of S-bounds always converge after a number of applications of the refinement routines, and if so, how quickly do they converge? Can we formulate rules for using bounds around arbitrary subtrees, rather than just on primitives? And as S-bounds are based on providing just an upper bound to the volume that a subtree influences, can we create a bounding scheme that is based on both upper and lower bounds?

12.2.5 Null Object Detection

I have used some reasonably simple heuristics for choosing and ordering the covering sets used in my NOD routine. Can we find better rules for this choice procedure? Would the use of procedures that combine results from non-neighbouring subtrees give better results, even if these routines have a greater than linear time complexity? Could we rewrite the CSG trees to form equivalent trees that would improve the efficiency of NOD routines?

12.2.6 Clash Detection

In this thesis I have tended to look at clash detection as an abstract problem. Can we create procedures that are experts at solving certain types of clash detection problems, such as those that have to be solved in assembly operations? Can we find solutions of practical clash detection problems that are significantly simpler than the general solutions that I have discussed? Can we build up a
library of clash detection macros that will enable a semi-autonomous system to solve most clash detection problems itself? If we are using a multiple interference detection algorithm can we use "temporal coherence" to create faster algorithms? And in the general case does the sweeping method (possibly using a boundary representation) have any advantages over four-dimensional interference detection?

12.2.7 Using Clash Detection In Other Procedures

For future applications it seems plausible that a procedure for performing clash detection could be used as a part of other procedures, such as a procedure for solving findpath, and this immediately raises two questions. Firstly, what information can be provided by a clash detection system to help a higher level system (e.g., time of first contact, approximate description of the clash hypervolume)? And secondly, can we define useful procedural interfaces between an "intelligent" geometric modelling system and other systems?

12.2.8 Statistical Analysis

Can we find statistical models that usefully describe "typical" objects, or "typical" interference detection situations, or "typical" clash detection situations, and that can be used to estimate or compare the computational efficiency of algorithms? What parameters could we use to characterise these situations? Can we take into account factors that defy the independence condition that I used in my analysis, such as spatial relationships? Can we provide meaningful statistical analyses of other algorithms, such as mindist() or algorithms for generating sweeps of objects? And can we find ways to analyse algorithms that use approximations or heuristics?
Appendix A: Set Theory

In this appendix I am concerned with establishing the set-theoretic results required for the development of, and extensions to, the r-set model in four dimensions; namely theorems 2.1, 2.2, 2.3 and 2.4. I also establish theorem 10.1, regarding the sweeping operator $Sw$.

Lemma A.1

For any set $S$ a point $x$ is in the closure of $S$ if for every open set $B$ with $x \in B$, $B \cap \text{in}(S) \neq \emptyset$.

Lemma A.2

For any set $S$,
\[
\text{in}(S) = \text{cl}(S).
\]

Lemma A.3

For any set $S$,
\[
\text{in}(S) \subset S \subset \text{cl}(S) \text{ and } \text{cl}(S) = \text{in}(S) \cup \text{by}(S).
\]

Lemma A.4

For any subset $S$ of a topological space $X$ the sets $\text{in}(S)$, $\text{by}(S)$, and $\text{in}(\overline{S})$ are mutually disjoint and fill the whole space, i.e.,
\[
X = \text{in}(S) \cup \text{by}(S) \cup \text{in}(\overline{S}).
\]
Lemmas A.1–A.4 are part of the folklore of topology, and proofs of these lemmas may be found in most of the standard texts on the subject.

**Lemma A.5**

A subset $S$ of a topological space is regular iff (a) $S$ is closed and (b) for every open $B$ such that $B \cap S \neq \emptyset$, $B \cap \text{int}(S) \neq \emptyset$.

**Proof**

Firstly assume that we are given a set $S$ satisfying (a) and (b). Then for any $x \in S$ and any open $B$ with $x \in B$ it is true that $B \cap S = \emptyset$, and so $B \cap \text{int}(S) = \emptyset$. But then $x \in \text{cl}\text{in}(S)$ (by Lemma A.1). So

$$S \subseteq \text{cl}\text{in}(S)$$

Conversely, if $x \in \overline{S}$, then as $S$ is closed $\overline{S}$ must be open and there exists an open $B$ with $x \in B$ and $B \subseteq \overline{S}$. So $B \cap \text{int}(S) = \emptyset$, and $x \notin \text{cl}\text{in}(S)$. Thus

$$\overline{S} \subseteq \text{cl}\text{in}(S)$$

Combining (1) and (2) shows that conditions (a) and (b) imply that $S$ is regular.

Now if $S$ is a regular set then it is certainly closed. Also for any open $B$ such that $B \cap S \neq \emptyset$ we can pick an $x$ from $B \cap S$. $S$ is the closure of $\text{int}(S)$, and so as $B$ is an open set around $x$, $x \in S$, $x \in B$ and $B \cap \text{int}(S) \neq \emptyset$ by the definition of closure. So the regularity of $S$ implies both (a) and (b).

QED
Lemma A.6

If $T$ and $V$ are topological spaces, and if $A$ is a closed subset of $T$, and $B$ is a closed subset of $V$, then $A \times B$ is a closed subset of $T \times V$ (under the usual product topology).

Proof

$A \times B$ is closed iff $\overline{A \times B}$ is open, and $\overline{A \times B}$ is the set of pairs with at least one element not in $A$ or not in $B$. So $\overline{A \times B} = \overline{A} \times \overline{V} \cup T \times \overline{B}$, and as all of these are open, so is $\overline{A \times B}$.

QED

A function that preserves open sets is called an open mapping. A function whose inverse is an open mapping is said to be continuous. Most of the functions that we are concerned with here are continuous open mappings.

Lemma A.7

If $f$ is a continuous, 1-1, onto and open mapping to a topological space $T$ then

$$\text{in} f = \text{f in}$$

Proof

$\text{in}(S) \subseteq S$ for all $S$, therefore $f \text{ in}(S) \subseteq f(S)$. Also as $\text{in}(S)$ is always open then so is $f \text{ in}(S)$, and further by applying $\text{in}$ to both sides of the relationship above we get

$$\text{in} f \text{ in}(S) = f \text{ in}(S) \subseteq \text{in} f(S)$$

(1)
To show the converse, let $x \in \text{inf}(S)$. Then there exists an open $B$ with $x \in B$ and $B \subseteq f(S)$. Therefore, as $f$ is continuous and onto, $f^{-1}(B)$ is open, and also as $f$ is 1-1,

$$f^{-1}(B) \subseteq f^{-1}(f(S)) = S.$$ 

In particular, $f^{-1}(x) \in f^{-1}(B)$, and so $f^{-1}(x) \in \text{in}(S)$. Now as $f$ is 1-1, $f^{-1}(x) \in \text{fain}(S)$, and so $x \in f^{-1}(S)$. Thus

$$\text{inf}(S) \subseteq f^{-1}(S) \quad (2)$$

and combining (1) and (2) gives

$$\text{inf} = f^{-1}\text{in}$$

QED

**Lemma A.8**

$$\text{in}(\text{clos}(\text{in}(S)) = \text{in}(S)$$

**Proof**

It suffices to prove that $\text{in}(\text{clos}(S)) = S$ for all open $S$.

$$\text{in}(\text{clos}(S)) = \text{in}(\text{in}(S) \cup \text{by}(S))$$

(Lemma A.3)

$$= \text{cl}(\text{in}(S) \cup \text{by}(S))$$

(Lemma A.2)

$$= \text{cl}(\text{in}(S))$$

(Lemma A.4)

$$= \text{in}(S) \cup \text{by}(S)$$

(Lemma A.3)

$$= \text{in}(S)$$

(Lemma A.4)

$$= S$$

QED
Lemma A.9

A continuous, 1-1, onto and open mapping preserves regularity.

Proof

Let $f$ be such a mapping from a topological space $T$ to a topological space $V$, and use the notion of regularity from Lemma A.5. If $A$ is a regular subset of $T$ then $\overline{A}$ is open in $T$, thus $f(\overline{A}) = \overline{f(A)}$ is open in $V$, and so $f(A)$ is closed in $V$.

If $B'$ is an open subset of $V$ with $B' \cap f(A) = \emptyset$ then $B = f^{-1}(B')$ is open in $T$ with $B \cap A = \emptyset$. So $B \cap \text{in}(A) = \emptyset$ (Lemma A.5), and there exists an $x$ such that $x \in B$ and $x \in \text{in}(A)$; thus $f(x) \in B'$ and $f(x) \in f(\text{in}(A)) = \text{in}(f(A))$ (by Lemma A.7). So $B' \cap \text{in}(A) = \emptyset$, and $f(A)$ is regular in $V$ (as it is also closed in $V$).

QED

Lemma A.10

If $T$ and $V$ are topological spaces, $C$ a regular subset of $T$, and $D$ a regular subset of $V$, then $C \times D$ is a regular subset of $T \times V$ (under the usual product topology).

Proof

Use the notion of regularity from Lemma A.5. $C \times D$ is closed (Lemma A.6). So if $B$ is an open subset of $T \times V$ such that $B \cap (C \times D) = \emptyset$ then there exist sets $F$ and $G$ such that $F$ is an open subset of $T$, $G$ is an open subset of $V$, and $B = F \times G$ (by the definition of the usual product topology). Then as $F \cap C = \emptyset$, and $C$ is regular, then $F \cap \text{in}(C) = \emptyset$. Similarly $G \cap \text{in}(D) = \emptyset$. So $(F \times G) \cap (\text{in}(C) \times \text{in}(D)) = \emptyset$, and as $(\text{in}(C) \times \text{in}(D)) \subseteq \text{in}(C \times D)$, then $B \cap \text{in}(C \times D) = \emptyset$. So $C \times D$ is regular in $T \times V$.

QED
Lemma A.11

If $f$ is any continuous, 1-1, onto and open mapping $f: T \rightarrow T$, where $T$ is a topological space with regular operator $rg$ (= $\text{cl} \circ \text{in}$), then

$f \circ rg = rg \circ f$

Proof

For all $S$,

\[
\text{in}([f \circ rg](S)) = \text{in} \circ f \circ rg(S)
\]

\[
= [f \circ \text{in} \circ \text{cl} \circ \text{in}](S) \quad \text{(Lemma A.7)}
\]

\[
= [f \circ \text{in}](S) \quad \text{([in] \circ \text{cl} \circ \text{in} = \text{in}, \text{Lemma A.8})}
\]

\[
= [\text{in} \circ f](S) \quad \text{(Lemma A.7)}
\]

Now we can apply $\text{cl}$ to both sides of this equation in turn.

For the left-hand side

\[
[\text{cl} \circ \text{in}][[f \circ rg](S)] = \text{rg}([f \circ rg](S))
\]

\[
= (f \circ rg)(S)
\]

as $\text{rg}(S)$ is regular and $f$ preserves regularity (Lemma A.9).

For the right-hand side

\[
[\text{cl} \circ \text{in} \circ f](S) = (\text{rg} \circ f)(S)
\]

Therefore

\[
[f \circ rg](S) = (\text{rg} \circ f)(S) \quad \text{for all } S
\]

$f \circ rg = \text{rg} \circ f$

QED
Lemma A.12

Let $F$ be a $1-1$ and onto mapping $F: T \to T$, $f$ a mapping $f: T \to \text{Bool}$ (where $\text{Bool}$ is the set of truth values, \{true,false\}), and $I^{-1}$ be the inverse indicator functional

$I^{-1}: (T \to \text{Bool}) \to \text{subsets of } T$ given by

$I^{-1}(g) = \{ x \in T \mid g(x) = \text{true} \}.$

Then

$I^{-1}(f \circ F^{-1}) = F \circ I^{-1}(f)$

Proof

$x \in I^{-1}(f \circ F^{-1}) \iff (f \circ F^{-1})(x) \\
\iff f(F^{-1}(x)) \\
\iff F^{-1}(x) \in I^{-1}(f) \\
\iff x \in F \circ I^{-1}(f)$

QED

The next definition is meant to capture precisely what we mean by a shape being represented by a regularised Boolean combination of half-space functions.

Definition A.13

If $f$ is a function $f: T \to \text{Bool}$ I define a functional

$\rho: (T \to \text{Bool}) \to T$ by $\rho = \text{rgI}^{-1}$

and say that $f$ represents $\rho(f)$. 

- 189 -
The next lemma uses the concepts of a location function, as defined in section 2.3, and the extrusion operator, as defined in section 2.4. Location functions return a transformation of three-dimensional space which is 1-1 and onto, and is also generally continuous.

Lemma A.14

$$\rho( f\circ L^{-1} ) = \text{Ex}(\rho(f),L)$$

where $L$ is a continuous location function.

Proof

For all $t$,

$$\rho( f\circ (L(t))^{-1} ) = (\text{rgo}(L(t))^{-1})(f\circ (L(t))^{-1})$$

$$= (\text{rgo}(L(t))\circ I^{-1})(f)$$

$$= (L(t)\circ \text{rgo})(I^{-1}(f))$$

$$= L(t)(\rho(f))$$

$$= \{ x \mid x \in L(t)(\rho(f)) \}$$

and so

$$\rho( f\circ L^{-1} ) = \{ (x,t) \mid x \in L(t)(\rho(f)) \} = \text{Ex}(\rho(f),L).$$

QED

Lemma A.15

If an object $R$ is given by $I^{-1}(f)$ for some Boolean function $f$, and $L$ is a continuous location function, then

$$\text{Ex}(R,L)$$

is given by $I^{-1}\circ f\circ L^{-1}$

Proof

$$x \in R \text{ iff } f(x) = \text{true}$$
Theorem 2.1

If TIMES is regular (in R), S is regular (in E3), and L continuous, then \( \text{Ex}(S, L) \) is regular (in E4).

Proof

First consider the subset of E4, SxTIMES. By Lemma A.10 this is regular, as it is the product of two regular sets. Now consider the function \( f: E4 \rightarrow E4 \) which is given by

\[ f(x,t) = (L(t)(x),t). \]

Then \( f \) is a continuous, 1-1, onto and open mapping, and so (by Lemma A.9) \( f(SxTIMES) = \text{Ex}(S, L) \) is regular.

QED

Theorem 2.2

If TIMES and S are closed and bounded, and L continuous, then \( \text{Ex}(S, L) \) is bounded.

Proof

Consider any \( x \in S \). Then as TIMES is closed and bounded and \( L \) is continuous then \( L(TIMES)(x) \) is bounded - by \( d_1 \) say. (This result follows from elementary analysis.) But for any \( y \in S \), \( |x-y| \) is bounded - by \( d_2 \) say. So \( L(TIMES)(S) \) is bounded by \( d_1 + d_2 \) (by the triangle inequality).
Theorem 2.3

If TIMES is a connected subset of \( \mathbb{R} \), \( S \) a connected subset of \( E^3 \), and \( L \) continuous, then \( Ex(S, L) \) is connected (in \( E^4 \)).

Proof

Let \((x, t)\) and \((y, u)\) be any two points in \( Ex(S, L) \). Assume without loss of generality that \( u < t \), and consider the path

\[ P_1 = (L(o)(L(u))^{-1}(y), o) \quad \text{for } o \in [u, t]. \]

Then as \( y \in L(u)(S) \), \( L(o)(L(u))^{-1}(y) \in L(o)(S) \), and so \( P_1 \) lies in \( Ex(S, L) \). So as \( L \) is continuous, \( P_1 \) is a valid arc. Furthermore, as \( S \) itself is connected, there exists an arc from the end of \( P_1 \) (i.e., \( L(t)(L(u))^{-1}(y) \)) to \( x \). Project this into \( Ex(S, L) \) by setting time to \( t \), and call this path \( P_2 \). Then \( P_1 \) followed by \( P_2 \) connects \((x, t)\) and \((y, u)\).

QED

Definition A.16

By a regularised Boolean function \( F \) of sets \( I \) mean the usual set-theoretic function

\[ F: T^n \rightarrow T \]

where \( T \) is some topological space.

We are interested in \( T \) as \( E^3 \), or \( T \) as \( E^4 \). In a CSG context \( F \) is defined by a CSG tree, and is regarded as a function from the primitives of the tree to the set represented by the tree.
Theorem 2.4 (Distribution Theorem)

Let R be a set given by a regularised Boolean function F on E3, so that

\[ R = F(p_1, p_2, \ldots, p_n) \]

Then if L is a continuous location function,

\[ \text{Ex}(R, L) = F(\text{Ex}(p_1, L), \text{Ex}(p_2, L), \ldots, \text{Ex}(p_n, L)) \]

Proof

By the usual folklore of CSG the primitives in a CSG representa-
tion can themselves be represented by indicator functions; so let
each \( p_i \) be given by \( p_i = I^{-1}(p_i) \). Then, also by the folklore, given
the function \( F \) there exists a function \( f \) with \( f : \text{Bool}^n \rightarrow \text{Bool} \) such
that

\[ f(p_1, p_2, \ldots, p_n) \text{ represents } R. \]

Now by Lemma A.15, \( \text{Ex}(p_i, L) \) is given by \( I^{-1}(p_i \circ L^{-1}) \) for each \( i \). So

\[ f(p_1 \circ L^{-1}, p_2 \circ L^{-1}, \ldots, p_n \circ L^{-1}) = f(p_1, p_2, \ldots, p_n) \circ L^{-1} \]

By Lemma A.14 the right hand side of this equation represents
\( \text{Ex}(R, L) \). But as \( f \) exactly mimics \( F \) and \( p_i \circ L^{-1} \) represents \( \text{Ex}(p_i, L) \),
the left hand side represents

\[ F(\text{Ex}(p_1, L), \text{Ex}(p_2, L), \ldots, \text{Ex}(p_n, L)) \]

So \( F(\text{Ex}(p_1, L), \text{Ex}(p_2, L), \ldots, \text{Ex}(p_n, L)) \) and \( \text{Ex}(R, L) \) are both represented
by the same Boolean function, and as \( \text{Ex}(R, L) \) is regular (Theorem 2.1)
both are regular and are thus the same set.

QED
Theorem 10.1

If $S$ is a regular set and $L$ is a continuous location function then $Sw(S, L)$ is a regular set.

Proof

Use the notion of regularity expressed in Lemma A.5. Let $S' = Sw(S, L)$; if $B$ is any open set such that $B \cap S' \neq \emptyset$ then there exists an $x$ in both $B$ and $S'$. As $x \in S'$ then there exist a $t$ and a $S''$ such that $x \in S'' = L(t)(S)$, and as $S$ is regular and $L(t)$ is continuous then $S''$ is regular; and so as $B \cap S'' \neq \emptyset$ (it contains $x$) then $B \cap \text{in}(S'') = \emptyset$. But $\text{in}(S') \subseteq \text{in}(S'')$, and so $B \cap \text{in}(S') = \emptyset$ as required.

It remains to show that $S'$ is closed. A set is closed iff it contains all its limit points. Let $x$ be any limit point of $S'$. So there exists a sequence of points $\{x_n\}$ and a sequence of times $\{t_n\}$ such that

$$\lim_{n \to \infty} L(t_n)(x_n) = x$$

Now by the Bolzano-Weierstrass Theorem, $\{t_n\}$ must contain a convergent sub-sequence. Say this sequence is $\{\tau_1\}$, and that the corresponding sub-sequence of points from $\{x_n\}$ is $\{y_1\}$. As the sequence $\{L(\tau_1)(y_1)\}$ converges, $L$ is continuous and $\{\tau_1\}$ converges, so must $\{y_1\}$ to $y$, say. So $\{L(t)(y_1)\}$ converges to $x$. But as TIMES is closed then $t \in$ TIMES, and as $S$ is closed $y \in S$. So $x \in L(t)(S) \leq S'$; and so $S'$ is closed, as required.
Appendix B: Hidden Line Removal

This appendix is concerned with one particular geometric function as an example of how algorithms can be designed. The example I have chosen is hidden-line removal, which is used to produce true-views of modelled objects. I have chosen this particular example for three reasons; firstly, because I hope that it is fairly easy for the reader to follow; secondly, because it happened to be an algorithm that was designed for robmod with the experience of the techniques described in chapter 4; and thirdly, because its development lead to the S-bound techniques described in chapter 6.

A simple form of hidden-line removal which is often available in modelling systems relies on the fact that if an edge is bounded by two faces that point away from the viewer then the edge cannot be visible. This produces a good result for simple objects, as shown in figure B.1a, but for more complex objects, or objects that are obscured by others, the method fails. Figures B.1b, B.1c, and B.1d show a more complex object, its wireframe, and the result of using the "local" hidden-line removal described.

A very simple algorithm for doing true hidden-line removal is as follows. Take the wireframe edges, and generate a larger set of edges by splitting all edges by any other edges that pass in front of them (The segments α and β are an example in figure B.1d). Then check to see if any of the edges are covered, and discard them if they are. This is a simple technique, but even in a boundary model it is quite expensive - $O(n^3)$ - as you have first to generate $O(n^2)$ edge segments, and then test each to see if they are covered. Many modelling systems use Warnock's algorithm, or something similar, for displaying static objects, as this does give a significant speed up over the basic method ([Warnock 69]). Warnock's algorithm has similarities with those which were later adopted by the group at Bath, in that the problem is solved through a set of windows which are dynamically chopped until the view through each is "simple enough".
However here it is a boundary representation that is being chopped, not a CSG representation.

One recent solution has been to use the ray-casting technique. However there is the problem that then we are trying to identify an infinitesimal set (the set of visible edges), with a finite number of rays, and so something like the 'flour bag' approach has to be used. As I was already using the simple Bath algorithms (for drawing
I looked closely at converting their algorithm for hidden-surface drawings; a hidden-surface drawing gives a shaded picture of an object. A slightly simplified description of the Bath group's algorithm follows; the real version uses several heuristics to speed up the process a little bit more ([Woodwark & Quinlan 82]).

The Bath Group's Hidden-Surface Algorithm

Recall that the basic Bath algorithm is based on a divide-and-conquer technique, whereby the working volume is split into octants, and each octant is drawn in turn. In the hidden-surface version, the idea is that the object is first transformed to fit into a box so that the observer appears to be looking straight at one side of the box (figure B.2). Then when the octant division takes place the octants that are closest to the observer are examined first, and if they cover their entire window then the octant behind them can be safely ignored. This effect is especially noticeable after several subdivisions. With the Bath group's program the way that this drawing is controlled is by using a data-structure called a quad-tree ([Hunter & Steiglitz 79]). Quad-trees were developed by the image-processing community as a way of compressing the information in a pattern.

As an example of a quad tree, consider the very simple pattern in figure B.3a; figure B.3b shows a form of the quad tree for this pattern. The way that this pattern is created is by a divide-and-conquer mechanism; that is, if a pattern is deemed too complicated to
be described directly it is split into four simpler patterns, and the process continues until either the minimum resolution is reached or the pattern has been described. In the Bath system the quad tree is used as a dynamic mask to describe the parts of the screen that are already covered; there is an obvious correspondence between the quads in the quad tree and a projection of the octants in the divide-and-conquer decomposition. There is also a correspondence between the quads and the "array fill" mechanisms provided with practically all graphical devices capable of producing shaded pictures.

However, when I looked at this algorithm with a view to doing hidden-line removal it became obvious that there were problems. In particular, the quad-tree gives a silhouette of the area of the screen covered so far, and it is not easy to augment it to produce edges. Instead I decided to try to abstract some of the ideas present in these algorithms and to design another one.

**My Hidden-Line Removal Algorithm**

Given the CSG description of the shape, I proceed as follows.

*Step 1*
The object is scaled and transformed to fit into a unit cube so that the observer appears to be looking at one face (the size of the object is established by using the S-bound mechanism that is described in chapter 6). Then use the Bath group's localisation mechanism to split the box into prisms, whose major axis is in the viewed direction. Call the plane of the face of the box that the observer sees the x-y plane, so that the z-direction is a vector pointing at the viewer (the situation is illustrated in figure B.4, with one of the prisms highlighted). Then the prisms are divided into both the x- and y- directions, until their contents (as given by the tree simplification routine) are simple enough for the next stage. (Typically my program divides one to three times at this stage.)

Step 2

We now have a set of prisms, and the total view is the sum of the views from each prism. Now for each prism do an "intelligent" z-division to get a sequence of interesting regions of space, as follows. From now on I shall use a two-dimensional example for simplicity; so consider the (simple) prism in figure B.5, where only the half-spaces left in the prism after tree simplification (localisation) are shown. Mark the z-extents of each primitive in the prism, and use these to build up a step-function which gives the maximum number of non-redundant primitives as a function of z (figure B.6), and thus a set of potential regions in which real line segments may
be found. The regions containing no primitives \((A,C,E,G)\) are of immediate interest because they are "constant", and we can do tree-simplification within them to get either \(\top\) or \(\bot\) as a result. If any such region gives \(\top\) then it forms a wall, and all subsequent regions must be hidden and can be forgotten. For example, if in figure B.6 region C simplified to \(\top\), then only the primitives in region B can be visible. Another fast test can be made with the \(1\)-primitive regions; they are either invisible, or hide subsequent regions.
In general we will end up with a set of complex regions containing at least two primitives; the only possibility for the example in figure B.6 is for region D to be declared complex, and then only if regions A and C simplify to \( \Box \). These are the regions that take the place of the octants in the simplified Bath algorithm. So now simplify the CSG tree for the prism w.r.t each region, and end up with an ordered set of regions and a CSG description valid for each one.

Step 3

Now consider solving the hidden-line problem for the first region. I use a generate and classify algorithm; the first set of candidate edges is generated just as for the partial wireframe in figure B.1. Each edge is a potentially visible edge (pv edge), and each pv edge is first split w.r.t all the other edges in front of it, to make a new set of pv edges. I now test to see if a pv edge is still potentially visible by casting a ray from the mid-point of each pv edge, back towards the observer. This uses Ray_cast() from section 4.3 and, as figure B.7 illustrates, my choice of the mid-points for the source of the ray means that we never have to worry about rays meeting two primitives simultaneously in that routine. For the first region, all pv edges that survive this stage are truly visible, and may be drawn.

![Figure B.7](image_url)
We can carry out a similar analysis for the other regions, and we end up with a list of pv edges, all of which would be visible if the observer looked at just that region, but they may be obscured by other regions. The Bath group use a single mask to decide on whether any feature is visible (the quad tree); I decided to simplify the algorithm by using a very simple set of masks, one for each possibly obscuring region. The final list of visible line-segments is kept for each region as each region is dealt with, working away from the observer. Note that any pv edges in one region can only be split by the visible edges in previous regions. Such a previous region splits the pv edge list that it is given, and applies Ray_cast() with its localised CSG description to discover which pv edges are visible through that region. This region then passes the new list of pv edges to its predecessor, and this process continues until the first region is passed. In effect, the CSG tree and list of visible edges in each region becomes the mask for that region.

Step 4

Draw all the visible edges in each region.

An algorithm like this should make it clear how tight the link can become between an efficient program and its data-structure. In practice the algorithm does work and, given the pre-existing tools and procedures, it was quite easy to implement.

Things to Note

The development of this algorithm raised several points. They are:

1) Consider the rectangle in figure B.8, which is being described by the intersection of four linear half-planes, labelled A, B, C and D. The effect of the simplification routine on the four half-planes within the prism shown is to leave them alone, and yet only two of the edges of the rectangle intersect the prism (this effect is not
nearly so noticeable with simplification regions that are nearly cubical). This observation lead me to develop the S-bound scheme of chapter 6, which allows such cases to be handled intelligently.

2) One problem with the mask scheme described is that we cannot detect a covered prism unless it happens to be covered by one region. For example, if the two regions of figure B.9 occur in a prism then between them they completely cover the prism, and so we need not con-

Figure B.9

tinue. I know of no easy answer to this problem, but note that this should only occur near complex parts of the object, that is, when many primitives from the CSG description come close together.

3) I could have treated the solution of the hidden-line problem in each region as another problem worthy of yet another divide-and-
conquer solution (but this has not been attempted in my present implementation of this algorithm).
Appendix C: robmod programs

This appendix contains three further examples of ROBMOD programs. The first example makes up the various bodies that compose the Puma robot model which is used in the workstation examples. The second example uses the body files that describe the parts of the Puma and links them together to form a single assembly for the whole robot. The third example uses the Puma assembly in a simple implementation of ClashAT(), the "Achilles and the Tortoise" multiple interference detection algorithm.

10 rem This program makes the various bodies that are used
20 rem in the Puma workstation model.
30 rem
40 rem The first set of objects are parts of the robot,
50 rem starting at the base and working towards the gripper.
60 rem
100 rem The base
110 b1 = cyl 6.8 0.95 rotx -90
120 rem a simple cylinder
130 shldr = cyl 3.5 0.95 roty 90 to -1.1 6.8 0
140 rem another cylinder
150 base = b1 + shldr
160 rem
200 rem The upper arm
210 a = cyl 1.04 2.2 + cube 4.32 3.04 1.04 to 0 -1.52 0
220 rem make something that is too big
230 b = a + cyl 1.04 0.8 to 4.32 0 0
240 c = ( phalfy * phalfy rotz -10 ) to 0.26 1.5 0
250 upperarm = ( b * ( c * c rotx 180 ) ) to 0 0 -0.52
260 rem and chop the excess off with a 'cheesewire'
270 rem
280 rem
300 rem The lower arm
310 d = cyl 0.86 1.17 + cube 4 1.6 0.86 to 0 -0.8 0
similar to the upperarm

e = phaly frtz -5 to 0 0.76 0

forearm = ( d * ( e * e rotx 180 ) ) to 0 0 -0.43

again, use a cheesewire

The wrist

wrist = cyl 0.86 0.41 to 0 0 -0.43

simple cylinder

The hand

parthand = cube 1 0.3 0.3 to 0.2 -0.15 -0.15

hand = parthand + cube 0.15 0.12 1 to 1.125 -0.06 -0.5

A finger

finger = cube 1 0.12 0.1 to 1.2 -0.06 0

only one is made, and this

is later copied

The fixed waist joint motor casing

motor = cyl 2 0.5 to -1.8 0 0 rotx -90

now make them into bodies, and save them

Base = cbody base

Upperarm = cbody upperarm

Forearm = cbody forearm

Wrist = cbody wrist

Hand = cbody hand

Finger = cbody finger

Motor = cbody motor

Base */u2/stephen/robmod/workshop/Base*

Upperarm */u2/stephen/robmod/workshop/Upperarm*

Forearm */u2/stephen/robmod/workshop/Forearm*

Wrist */u2/stephen/robmod/workshop/Wrist*

Hand */u2/stephen/robmod/workshop/Hand*

Finger */u2/stephen/robmod/workshop/Finger*
A Resonably Complicated Model of Peggy,
the Edinburgh Puma

waist =  60
shoulder = -120
elbow =   0
bend =    0
swivel =   50
tw =      105

default values for the joint angles

The hand is opened so the distance between the
fingers is gap, where 0 <= gap <= 0.7

The next few lines define the transformation chains
necessary to align the joint axes, and to apply the
required rotation about each axis.
"trmn" is the chain to move from the coordinate
system of link m to that of link n.

tr01 = roty waist
tr12 = rotz (shoulder+90) roty -90 to 2.48 6.8 0
tr23 = rotz elbow to 4.32 0 0.95
tr34 = rotx bend to 4.3 0 0
270 rem wrist assembly to forearm
280 tr45 = rotx tw rotz swivel
290 rem hand to wrist
292 rem
295 rem Now read in the stored body description files
297 rem
300 Base = rbody "/u2/stephen/robmod/workshop/Base"
320 Upperarm = rbody "/u2/stephen/robmod/workshop/Upperarm"
330 Forearm = rbody "/u2/stephen/robmod/workshop/Forearm"
340 Wrist = rbody "/u2/stephen/robmod/workshop/Wrist"
350 Hand = rbody "/u2/stephen/robmod/workshop/Hand"
360 Finger = rbody "/u2/stephen/robmod/workshop/Finger"
370 Motor = rbody "/u2/stephen/robmod/workshop/Motor"
400 rem
402 rem Make up an assembly of one of the fingers,
404 rem translated so that the fingers appear the
405 rem correct distance apart.
406 rem The complete gripper ("hand") will require two
408 rem of these, one mirrored about the x-y plane.
410 rem
420 tempfing = Finger to 0 0 (gap/2)
500 rem
502 rem Now to assembly the robot. The "\" at the
504 rem physical end of a line just serves as a
506 rem line continuation character.
508 rem
520 peggy = Motor @ (Base @ [Upperarm @ (Forearm @ [wrist @ 
(Hand @ tempfing @ tempfing rotx 180) \ 
tr45) tr34) tr23) tr12) tr01

- 208 -
An implementation of ClashAT(j in ROBMOD

(uses the Puma workstation model)

First define various parameters

maxspeed = 3

value "returned" by maxspeed()

ts = 0

initial value of time

tf = 5

maximum value of time

deltad = 0.1

minimum "depth" of clash to be noticed

for the puma model, 1 unit is 100mm

deltat = 0

minimum time of clash to be noticed

values for the robot joints as functions of time

(other joints left at their default values)

elbow = - 25 + (t*10)

elbow joint

waist = -(t*10) + 60

waist joint

n = 0

parameter to change the shoulder joint function

shoulder = -120 + (t*n)

shoulder joint

ClashAT(j itself

set the initial time

now for the main loop

if time is too large, exit the loop

calculate the minimum distance
640 dist = eval mindist( peggy, objs)
650 rem objs is an assembly, defined to be the obstacle(s)
660 if dist<=0 goto 900
670 rem if clashing, exit loop
680 t = eval t + ((dist +deltad)/maxspeed) +deltat
690 rem calculate new value of time
700 goto 600
800 rem Exit point for "no clash"
810 print "No Clash\n"
820 stop
900 rem Exit point for "clash"
910 print "Clash\n"
920 stop
Appendix D: CSG Trees and S-Bounds

This appendix is concerned with the results required for the chapters on CSG trees, S-bounds, and null object detection (Chapters 6 and 7).

**Structural Induction**

Structural induction is a proof method which is naturally applicable to trees, and it is a method that is discussed in many works, for example [Burstall 69]. Many of the proofs that follow use the method, and for completeness I quote here a version of the principle of structural induction.

**Principle D.1**

Let Pr be a predicate on CSG trees such that Pr(T) iff some property holds for T. Then if

(a) Pr(P) for all primitives P, and

(b) Pr(A) and Pr(B) implies Pr(A<op>B) for all tree operations <op>

then Pr(T) for all trees T.

So to use structural induction we first prove that the property holds for primitives in a CSG tree, then prove that the property is propagated to parents of subtrees with that property, and then we can deduce that the property holds for all trees. As usual we can assume, if it is necessary, that we are working under a fixed interpretation of the tree for a particular proof; also we can take the word "primitive" to mean the deepest entities in the CSG tree that we wish to consider. I present a proof that the method is sound which is itself based on a more familiar induction principle, namely
that of numerical induction, and any reader who is unhappy with structural induction may like to use the proof I give as a "macro" which can be used to expand any of my proofs that use structural induction into ones that use numerical induction.

Proof (of Principle D.1)
The proof runs by numerical induction on the size of CSG trees. Let the maximum depth of any tree T be denoted by d(T). d(T) is zero iff T is a primitive, and by condition (a) Pr(T) is true. So assume that d(T)<k implies Pr(T). Then if we are given a tree T with d(T)=k then it is of the form T_1<op>T_2. But then d(T_1)<k and d(T_2)<k which implies Pr(T_1) and Pr(T_2), and by condition (b) we have Pr(T_1) and Pr(T_2) implies Pr(T_1<op>T_2), i.e. Pr(T). So Pr(T), and by numerical induction on d(T), Pr(T) for all T.

QED

For convenience in the proofs that follow I will define an auxiliary function,

\[ \phi: \text{Bounded Trees } \rightarrow \text{Sets} \]

by the following rules:

1. If T is a primitive then
   \[ \phi_B(T) = i(T) \]

2. If T is a tree of the form AUB then
   \[ \phi_B(T) = ( I_B(A)U I_B(B) ) \]

3. If T is a tree of the form A\&B then
   \[ \phi_B(T) = ( I_B(A)\& I_B(B) ) \]

4. If T is a tree of the form A/B then
   \[ \phi_B(T) = ( I_B(A)/I_B(B) ) \]

Then it is an easy proof (by structural induction) that I_B(T) =
Lemma D.2

Let $T$ be a bounded tree with bounding function $\beta$ and interpretation of primitives $i$. If $S$ is an arbitrary set and $i'$ is the interpretation given by

$$i'(p) = \begin{cases} i(p)AS & \text{if } p \text{ is a zero-order primitive of } T \\ i(p) & \text{otherwise} \end{cases}$$

then $I'_\beta(T) = I_\beta(T)AS$.

Proof

Structural induction on $T$. If $T$ is a primitive then

$$I'_\beta(T) = i'(T)AS(T) = i(T)ASAS(T) = I_\beta(T)AS$$
as required.

Otherwise if $T = T_1 <op> T_2$, where $<op>$ is one of $U$ or $A$, then if the property holds for $T_1$ and $T_2$

$$I'_\beta(T) = \{ I'_\beta(T_1) <op> I'_\beta(T_2) \} AS(T)$$

by associativity or distributivity

$$= I_\beta(T)AS$$
as required.

Finally, if $T = T_1 / T_2$ then all of the primitives in $T_2$ have non-zero-order in $T$, and so

$$I'_\beta(T) = \{ I'_\beta(T_1) / I'_\beta(T_2) \} AS(T)$$

$$= \{ I_\beta(T_1)AS / I_\beta(T_2) \} AS(T)$$

$$= I_\beta(T)AS$$
as required.
Lemma D.3

Let \( \Psi \) be a bounded tree with two bounding functions \( \beta' \) and \( \beta'' \), and also with two interpretations, \( i' \) and \( i'' \). If we are given a subtree \( T' \) of \( \Psi \), and a subtree \( T'' \) of \( T' \) such that

\[ i'(P) = i''(P) \]

for all primitives \( P \) which are not primitives of \( T'' \), and

\[ I_{\beta'}(T'') \preceq I_{\beta''}(T'') \]

then if \( T'' \) is positive in \( T' \), \( I_{\beta'}(T') \preceq I_{\beta''}(T') \);
else if \( T'' \) is negative in \( T' \), \( I_{\beta'}(T') \succeq I_{\beta''}(T') \).

The theorem may be paraphrased by "making a positive subtree larger results in a larger tree, and making a negative subtree larger results in a smaller tree".

Proof

Use structural induction on \( T' \). If \( T'' \) is \( T' \), then the result holds trivially. Otherwise \( T' \) is of the form \( T_1 <\text{op}> T_2 \), for some set operation \( <\text{op}> \), and \( T'' \) is a subtree of either \( T_1 \) or \( T_2 \). Define the set order relation \( <\text{rel}> \) to be \( \preceq \) if \( T'' \) is positive in \( T' \), and \( \succeq \) otherwise. I need to show that

\[ I_{\beta'}(T') <\text{rel}> I_{\beta''}(T') \]

and I proceed by considering the cases.

If \( <\text{op}> \) is \( U \) or \( A \), assume without loss of generality that \( T' \) is a subtree of \( T_1 \). Then if \( <\text{op}> \) is \( U \) or \( A \), or if \( <\text{op}> \) is \( \cap \) and \( T'' \) is a subtree of \( T'_1 \) we have

\[ I_{\beta'}(T_1) <\text{rel}> I_{\beta''}(T_1) \]
and
\[ I_{\beta'}(T_2) = I_{\beta''}(T_2) \]
and so

\[ I'_{B'}(T') = \{ I'_{B'}(T_1') <op> I'_{B'}(T_2) \} \land B(T') \]
\[ \land \{ I''_{B''}(T_1') <op> I''_{B''}(T_2) \} \land B(T') \]

as required.

Otherwise \( T'' \) is a subtree of \( T_2 \) in \( T' = T_1/T_2 \), and so the sign of \( T'' \) in \( T_2 \) is the opposite of the sign of \( T'' \) in \( T' \). Thus \( I''_{B''}(T_2) <rel> I'_{B'}(T_2), I''_{B''}(T_1) = I'_{B'}(T_1), \) and

\[ I'_{B'}(T') = \{ I'_{B'}(T_1')/I'_{B'}(T_2) \} \land B(T') \]
\[ \land \{ I''_{B''}(T_1')/I''_{B''}(T_2) \} \land B(T') \]

as required. QED

Lemma D.4

If \( T' \) is a subtree of a bounded tree \( Y \), and \( T \) is a subtree of \( T' \), then

\[ I_B(T') = \{ I_B[T\emptyset](T')/B(T) \} \cup \{ I_B[T\emptyset](T')/B(T) \} \]

Proof

For the purpose of this proof I shall abbreviate \( I_{B[T\emptyset]} \) and \( I_{B[T\emptyset]} \) to \( I_\emptyset \) and \( I_\emptyset \) respectively. The proof is by structural induction on the composition of \( T' \). If \( T' \) is \( T \) then

\[ I_B(T') = \phi_B(T) \land B(T) \]
\[ = \{ [\phi_B(T)/B(T)] \cup [\phi_B(T)/B(T)] \} \]
\[ = \{ I_\emptyset(T')/B(T) \} \cup \{ I_\emptyset(T')/B(T) \} \]

as required.

Otherwise \( T \) is a proper subtree of \( T' \), and I proceed by cases.

Case 1: \( T' \) is \( T_1 \land T_2 \)

Assume without loss of generality that \( T \) is a subtree of \( T_1 \). Then
\[ \phi_{\beta}(T') = I_{\beta}(T_1) \land I_{\beta}(T_2) \]
\[ = \{ I_{\phi}(T_1) / \beta(T) \} U \{ I_{\lambda}(T_1) \lambda \beta(T) \} \land I_{\beta}(T_2) \]
by the induction hypothesis
\[ = \{ I_{\phi}(T_1) / \beta(T) \} \land I_{\beta}(T_2) \} U \{ I_{\lambda}(T_1) \lambda \beta(T) \} \land I_{\beta}(T_2) \}
by distributivity
\[ = \{ I_{\phi}(T_1) \lambda I_{\beta}(T_2) \} / \beta(T) \} U \{ I_{\lambda}(T_1) \lambda I_{\beta}(T_2) \} \lambda \beta(T) \}
Thus
\[ I_{\beta}(T') = \phi_{\beta}(T') \lambda \beta(T') \]
\[ = \{ I_{\phi}(T_1) \lambda I_{\beta}(T_2) \} / \beta(T) \} \land \beta(T') \} \]
by the induction hypothesis
\[ = \{ I_{\phi}(T_1) / \beta(T) \} U \{ I_{\lambda}(T_1) \lambda \beta(T) \} \}
\[ = \{ I_{\phi}(T_1) \} \land \beta(T) \} U \{ I_{\lambda}(T_1) \} \lambda \beta(T) \}
Thus
\[ I_{\beta}(T') = \phi_{\beta}(T') \lambda \beta(T') \]
\[ = \{ I_{\phi}(T_1) \} \land \beta(T') \} \]
by the induction hypothesis
\[ = \{ I_{\phi}(T_1) \} \land \beta(T') \} \land \beta(T) \}
Thus
\[ I_{\beta}(T') = \phi_{\beta}(T') \lambda \beta(T') \]
\[ = \{ I_{\phi}(T_1) \} \land \beta(T') \} \]
by the induction hypothesis

Case 2: T' is T \cup T_2
Assume without loss of generality that T is a subtree of T_1. Then
\[ \phi_{\beta}(T') = I_{\beta}(T_1) \land I_{\beta}(T_2) \]
\[ = \{ I_{\phi}(T_1) / \beta(T) \} U \{ I_{\lambda}(T_1) \lambda \beta(T) \} \land I_{\beta}(T_2) \]
by the induction hypothesis
\[ = \{ I_{\phi}(T_1) / \beta(T) \} U \{ I_{\lambda}(T_1) \lambda \beta(T) \} U \}
\[ = \{ I_{\phi}(T_1) \lambda I_{\beta}(T_2) \} / \beta(T) \} U \{ I_{\lambda}(T_1) \lambda I_{\beta}(T_2) \} \lambda \beta(T) \}
Thus
\[ I_{\beta}(T') = \phi_{\beta}(T') \lambda \beta(T') \]
\[ = \{ I_{\phi}(T_1) \} \land \beta(T') \} \]
by the induction hypothesis
\[ = \{ I_{\phi}(T_1) \} \land \beta(T') \} \land \beta(T) \}
Thus
\[ I_{\beta}(T') = \phi_{\beta}(T') \lambda \beta(T') \]
\[ = \{ I_{\phi}(T_1) \} \land \beta(T') \} \land \beta(T) \}
as required.

Case 3: T' is T_1 / T_2
If T is a subtree of T_1 then the result holds by case 1. So assume that T is a subtree of T_2. Then for any bounding function \beta
\[ I_\beta(T') = \{ I_\beta(T_1)/I_\beta(T_2) \} \land I(T') \]
\[ = \{ I_\beta(T_1) \land \hat{s}(T') \} / I_\beta(T_2) \]

i.e. \( I_\beta(T') = A/I_\beta(T_2) \) \hfill ...(1)
say, where \( A = I_\beta(T_1) \land \hat{s}(T') \).

Now

\[
\begin{align*}
[I_\emptyset(T')/\beta(T)] & \cup [I_\emptyset(T') \land \hat{s}(T)] \\
& = \{ A/I_\emptyset(T_2) \} / \beta(T) \cup \{ I_\emptyset(T_2) \}[A \land \hat{s}(T)] \\
& \quad \text{using (1) with } I_\emptyset \text{ and } I_\emptyset \\
& = \{ A/I_\emptyset(T_2) \land \hat{s}(T) \} \cup \{ A[I_\emptyset(T_2) \land \hat{s}(T)] \}
\end{align*}
\]

by the identities \((X/Y)Z = X/(YZ) \) and \((X/Y)Z = (XZ)/Y \).

\[
\begin{align*}
& = \{ A[I_\emptyset(T_2) \land \hat{s}(T)] \} \cup \{ A[I_\emptyset(T_2) \land \hat{s}(T)] \}
\end{align*}
\]

say

where \( B = I_\emptyset(T_2), C = I_\emptyset(T_2), D = \beta(T) \)

\[
\begin{align*}
& = \{ A[I_\emptyset(T_2) \land \hat{s}(T)] \} \cup \{ A[I_\emptyset(T_2) \land \hat{s}(T)] \}
\end{align*}
\]

\( A \land \{ [BUD] \cup [DAC] \} \)

\( A \land \{ [BUD] \cup [DAC] \} \)

\( A \land \{ [BUD] \cup [DAC] \} \)

Comparing (1) and (2), we see that the result will be true if

\[
\begin{align*}
I_\beta(T_2) &= \{ I_\emptyset(T_2) / \beta(T) \} \cup \{ I_\emptyset(T_2) \land \hat{s}(T) \} \\
& = \{ I_\emptyset(T_2) \} / \beta(T) \cup \{ I_\emptyset(T_2) \land \hat{s}(T) \}
\end{align*}
\]

Consider the left hand side of this; by the induction hypothesis

\[
\begin{align*}
I_\beta(T_2) &= \{ I_\emptyset(T_2) / \beta(T) \} \cup \{ I_\emptyset(T_2) \land \hat{s}(T) \} \\
& = \{ I_\emptyset(T_2) \} / \beta(T) \cup \{ I_\emptyset(T_2) \land \hat{s}(T) \}
\end{align*}
\]

\[
\begin{align*}
& = \{ I_\emptyset(T_2) \} / \beta(T) \cup \{ I_\emptyset(T_2) \land \hat{s}(T) \}
\end{align*}
\]

\( \text{as required.} \)

QED
Theorem 6.4
Let \( \mathcal{Y} \) be a bounded tree with a bounding function \( \beta \) that satisfies
\[
\begin{align*}
\beta(T) &> i(T) & \text{if } T \text{ is a primitive of } \mathcal{Y}, \text{ and} \\
\beta(T) &\leq \Omega & \text{otherwise.}
\end{align*}
\]

Then \( \beta \) is an S-bound function on \( \mathcal{Y} \).

Proof
It is sufficient to show that
\[
I'(T) = I'(T)
\]
for all bounding functions \( \beta' \) satisfying \( \beta' \upharpoonright \beta \), all interpretations \( i' \) satisfying \( i' \upharpoonright i \), and all subtrees \( T \) of \( \mathcal{Y} \); the proof is by structural induction on the composition of \( T \). Note that for all \( T \)
\[
I'(T) \leq \beta(T) \leq \beta'(T),
\]
as it is true by construction if \( T \) is a primitive, and it is trivially true for all other \( T \) as \( \beta(T) = \Omega \). Thus if \( T \) is a primitive then
\[
I'_\beta(T) = I'(T) \beta'(T) = I'(T),
\]
and the result holds. If \( T \) is not a primitive then \( \phi'_\beta(T) = I'(T) \) by the induction hypothesis and the definition of \( \phi \), and so
\[
I'_\beta(T) = \phi'_\beta(T) \beta'(T) = I'(T),
\]
as required. Therefore the result holds for all subtrees of \( \mathcal{Y} \), and in particular for \( \mathcal{Y} \) itself.

QED

Theorem 6.5 (The Upward Theorem)
Let \( \mathcal{Y} \) be a bounded tree with S-bound function \( \beta \). If \( T \) is any subtree of \( \mathcal{Y} \) with immediate subtrees \( T_1 \) and \( T_2 \) then another S-bound function for \( \mathcal{Y} \) is given by \( \beta' \), where
\[ \beta' = \beta[T \# S \beta(T)] \]

and the set \( S \) is given by

\[
S = \begin{cases} 
\beta(T_1) \cup \beta(T_2) & \text{if } T = T_1 \cup T_2 \\
\beta(T_1) \cup S(T_2) & \text{if } T = T_1 \land T_2 \\
\beta(T_1) & \text{if } T = T_1 / T_2 
\end{cases}
\]

Proof

It is sufficient to show that \( I'_{\beta''}(T) = I'(T) \) for any bounding function \( \beta'' \) with \( \beta'' \parallel \beta' \), where \( \beta' \) is defined as above, and any interpretation \( i' \parallel i \). The idea of the proof is that if \( \beta'' \) is any such bounding function then we can find another bounding function \( \beta^o \) satisfying the conditions

(a) \( I'_{\beta''}(T) = I'_{\beta^o}(T) \)

(b) \( \beta^o \) is identically equal to \( \beta'' \) except that \( \beta^o(T) \leq \beta(T) \), \( \beta^o(T_1) \leq \beta(T_1) \), and \( \beta^o(T_2) \leq \beta(T_2) \).

Condition (b) implies that \( \beta^o \) is itself an \( S \)-bound function (as \( \beta \) is), and as \( \beta^o \) is equal to \( \beta'' \) except on \( T \), \( T_1 \) and \( T_2 \) then it follows (from condition (1)) that \( I'_{\beta^o}(\psi) = I'_{\beta''}(\psi) \), and so both are equal to \( I'(\psi) \).

To choose \( \beta^o \) I consider the three cases corresponding to the operator of \( T \).

Case 1: operator is \( U \)

Consider the sets

\[
A = \beta''(T) \cup \beta(T) \quad \text{and} \\
B = \beta''(T) \cup \beta(T_1) \cup \beta(T_2).
\]

Then \( A \geq \beta(T) \), \( B \geq \beta(T_1) \), \( B \geq \beta(T_2) \), and

\[
A \# B = \beta''(T) \cup \{ \beta(T) \mid \beta(T_1) \cup \beta(T_2) \} = \beta''(T) \cup \beta'(T)
\]
Thus

\[
I_{\beta^m_n}(T) = \phi_{\beta^m_n}(T) \land \beta^m(T)
\]

\[
= \{ \phi_{\beta^m_n}(T_1) \land \beta^m(T_1) \} \cup \{ \phi_{\beta^m_n}(T_2) \land \beta^m(T_2) \} \} \land (A \land B)
\]

\[
= \{ \phi_{\beta^m_n}(T_1) \land \beta^m(T_1) \land \beta(T_1) \} \cup \{ \phi_{\beta^m_n}(T_2) \land \beta^m(T_2) \land \beta(T_2) \} \} \land A
\]

So if we take \( \beta^0(T) = A, \beta^0(T_1) = \beta^m(T_1) \land \beta, \) and \( \beta^0(T_2) = \beta^m(T_2) \land \beta \) then \( I_{\beta_0}(T) \) will be identically equal to the expression above and \( \beta^0 \) will be of the required form.

Operator is \( \land \)

Consider the sets

\[
A = \beta^m(T) \cup \beta(T),
\]

\[
B = \beta^m(T) \cup \beta(T_1), \text{ and}
\]

\[
C = \beta^m(T) \cup \beta(T_2).
\]

Then \( A \geq \beta(T), B \geq \beta(T_1), \) and \( C \geq \beta(T_2), \) and

\[
A \land B \land C = \beta^m(T) \cup \{ \beta(T) \land \beta(T_1) \land \beta(T_2) \}
\]

\[
= \beta^m(T) \lor \beta'(T)
\]

Thus

\[
I_{\beta^m_n}(T) = \{ \phi_{\beta^m_n}(T_1) \land \beta^m(T_1) \} \land \{ \phi_{\beta^m_n}(T_2) \land \beta^m(T_2) \} \land (A \land B \land C)
\]

\[
= \{ \phi_{\beta^m_n}(T_1) \land \beta^m(T_1) \land \beta(T_1) \} \cup \{ \phi_{\beta^m_n}(T_2) \land \beta^m(T_2) \land \beta(T_2) \} \} \land A
\]

and so if we choose \( \beta^0(T) = A, \beta^0(T_1) = \beta^m(T_1) \land \beta, \) and \( \beta^0(T_2) = \beta^m(T_2) \land \beta \) then \( I_{\beta_0}(T) \) will be identically equal to the expression above and \( \beta^0 \) will be of the required form.

Operator is \( \land \)

Consider the sets
\[ A = \beta^n(T) U \beta(T) \quad \text{and} \quad B = \beta^n(T) U \beta(T_1). \]

Then \( A \geq \beta(T), \ B \geq \beta(T_1), \) and \( AAB = \beta^n(T). \) So

\[
I'_{\beta^n}(T) = \{ \phi_{\beta^n}(T_1) A \beta^n(T_1) \} / \{ \phi_{\beta^n}(T_2) A \beta^n(T_2) \} \quad \text{A} (AAB)
\]

\[
= \{ \phi_{\beta^n}(T_1) A \beta^n(T_1) A \beta(T_1) \} / \{ \phi_{\beta^n}(T_2) A \beta^n(T_2) \} \quad \text{A} A
\]

and so if we choose \( \beta^0(T) = A, \beta^0(T_1) = \beta^n(T_1) A \beta(T_1) \) and \( \beta^0(T_2) = \beta^n(T_2) \)

then \( I'_{\beta^0}(T) \) will be identically equal to the expression above and \( \beta^0 \)

will be of the required form.

QED

**Theorem 6.6 (The Downward Theorem)**

Let \( \psi \) be a bounded tree with S-bound function \( \beta \). If \( T \) is any

subtree of \( \psi \), and \( T' \) is an immediate subtree of \( T \), then another

S-bound function for \( \psi \) is given by \( \beta' \), where

\[ \beta' = \beta[T' \# \beta(T) A \beta(T') \]

Proof

It is sufficient to prove that

\[ I'_{\beta^n}(\psi) = I'(\psi) \]

for any bounding function \( \beta^n \uparrow \beta' \), and any interpretation \( i' \sqsubseteq i \). I

will proceed by considering the cases of the set operation at \( T \), but

first, consider the sets

\[ A = \beta^n(T') U \beta(T) \quad \text{and} \quad B = \beta^n(T') U \beta(T'). \]

Then \( A \geq \beta(T), \ B \geq \beta(T'), \) and

\[
AAB = \beta^n(T') U [ \beta(T) A \beta(T') ]
\]

\[ = \beta^n(T') U \beta'(T') \]

\[ = \beta^n(T') \quad \text{as} \beta^n(T') \geq \beta'(T'). \]
Case 1: operator is $A$ 
Assume without loss of generality that $T = T'\lambda T''$. Then

$$I_{\beta''}^n(T) = \{ [\phi_{\beta''}^n(T')\lambda AAB]_A I_{\beta''}^n(T'') \} \cup I_{\beta''}(T)$$

$$I_{\beta''}^n(T) = \{ [\phi_{\beta''}^n(T')\lambda AB]_A I_{\beta''}^n(T'') \} \cup I_{\beta''}(T)$$

So consider the bounding function

$$\beta^o = \beta''[T\#(T')\lambda A; T'\#B].$$

Then $\beta^o \perp \beta$ (and so is an $S$-bound function), $I_{\beta''}^n(T) = I_{\beta^o}(T)$, and $\beta^o$ is identical to $\beta''$ outside $T$. So

$$I_{\beta''}^n(\psi) = I_{\beta^o}(\psi) = I'_{\psi}$$

as required.

Case 2: operator is $U$ 
Assume without loss of generality that $T = T'\Upsilon T''$, and note that for any bounding function $\beta$

$$I_{\beta'}^n(T) = \{ \phi_{\beta'}^n(T')\lambda \Upsilon(T')\lambda \Upsilon(T) \} \cup \{ I_{\beta'}^n(T''\lambda \Upsilon(T) \} \ldots (1)$$

Then consider the bounding functions

$$\beta^o = \beta''[T'\#B]$$

and

$$\beta^1 = \beta''[T'\#A\beta''(T)]$$

where $A$ and $B$ are defined above. Then $\beta^o \perp \beta$ and $\beta^1 \perp \beta$, and so both are $S$-bound functions. Using identity (1), and noting that $\phi_{\beta''}^n(T') = \phi_{\beta^o}^n(T') = \phi_{\beta^1}^n(T')$, we have

$$I_{\beta^o}^n(T) = \{ \phi_{\beta''}^n(T')\lambda AAB\beta''(T) \} \cup \{ I_{\beta''}^n(T'')\lambda A\beta''(T) \}$$

$$I_{\beta''}^n(T) = \{ \phi_{\beta''}^n(T')\lambda AAB\beta''(T) \} \cup \{ I_{\beta''}^n(T'')\lambda A\beta''(T) \}$$

$$I_{\beta^1}^n(T) = \{ \phi_{\beta''}^n(T')\lambda A \beta''(T) \} \cup \{ I_{\beta''}^n(T'')\lambda A\beta''(T) \}$$

and so

$$I_{\beta^o}^n(T) \leq I_{\beta''}^n(T) \leq I_{\beta^1}^n(T) \quad \ldots (2)$$
Using lemma D.3, and defining $\langle \text{rel} \rangle$ to be `< if $T$ is positive in $\psi$, > otherwise, we have

$$I'\beta'(\psi) \langle \text{rel} \rangle I'_\beta \psi \langle \text{rel} \rangle I'_\beta \psi \langle \text{rel} \rangle I'_\beta \psi \ ... (3)$$

But $\beta^0$ and $\beta^1$ are both S-bound functions, and so the outermost terms in (3) are both equal to $I'(\psi)$, and

$$I'_\beta \psi = I'(\psi)$$

as required.

**Case 3: operator is /**

If $T$ can be written in the form $T'/T''$ then the result holds as in case 1, as $T = T'/T''$. So assume without loss of generality that $T = T'/T''$. This case is by far the most difficult case to prove.

Firstly, define the extra bounding functions

$$\beta^0 = \beta'[T\#\beta(T)] \quad \text{and} \quad \beta^1 = \beta'[T\#\beta'[T']U\beta(T')]$$

Then note that $\beta^1 \perp \beta$, and is thus an S-bound function. Secondly, define the interpretation

$$i''(p) = \begin{cases} i'(p)A\beta(T) & \text{if } P \text{ is a primitive of zero-order in } T' \\ i'(p) & \text{otherwise} \end{cases}$$

and note that by lemma D.2

$$I'(T') = I'(T') A \beta(T) \quad ... (4)$$

The main stages of the proof follow.

**Step 1**

Consider $\beta^0$, and note that

$$I'_\beta \psi(T) = \{ I'_\beta \psi(T') / [\phi'_\beta(T')A\beta^0(T')] \} A \beta^0(T)$$

$$= \{ I'_\beta \psi(T') / [\phi'_\beta(T')A\beta'(T')] \} A \beta(T)$$

by the definition of $\beta^0$

$$= [I'_\beta \psi(T')A\beta(T)] / [\phi'_\beta(T')A\beta]$$

$$= \{ [I'_\beta \psi(T')A\beta(T)] / [\phi'_\beta(T')A\beta] \} U \{ [I'_\beta \psi(T')A\beta(T)] / A \}$$
by the identity \( X/(YZ) = (X/Y)U(X/Z) \)
\[ = \left[ I_{B}^{n}(T)\chi B(T) \right] / [\phi_{B}^{n}(T')\chi B] \]
as \( I_{B}^{n}(T')\chi B(T) \leq B(T) \leq A \)
\[ = I_{B}^{n}[T',\#B](T) \]

But \( B^{o}[T',\#B] \) \( B \), and so is an \( S \)-bound function, and \( B^{o}[T',\#B] \) is identical to \( B^{o} \) outside \( T \), and so
\[ I_{B}^{o}(\psi) = I_{B}^{o}[T',\#B](\psi) = I'(\psi) \]
i.e. \( I_{B}^{o}(\psi) = I'(\psi) \) \( \ldots (5) \)

Also note that the derivation of this equation will work under any interpretation \( i^{o} \subseteq i \), and in particular
\[ I_{B}^{n}(\psi) = I'(\psi) \] \( \ldots (6) \)

Step 2
Note that
\[ I_{B}^{n}(T) = \left\{ I_{B}^{n}(T')/I_{B}^{n}(T') \right\} \wedge B(T) \]
\[ = \left\{ I_{B}^{n}(T')/\left[I_{B}^{n}(T')\chi B(T) \right] \right\} \wedge B(T) \]
by the identity \( (X/Y)AZ = (X/(YZ))AZ \)
\[ = \left\{ I_{B}^{n}(T')/I_{B}^{n}(T') \right\} \wedge B(T) \]
by \( (4) \)
\[ = I_{B}^{n}(T) \]
as \( i''(P) = i'(P) \) for primitives of \( T'' \)

But \( i''(P) = i'(P) \) for all primitives which are not primitives of \( T \), and so
\[ I_{B}^{o}(\psi) = I_{B}^{n}(\psi) \] \( \ldots (7) \)

Combining \( (5), (6) \) and \( (7) \) gives
\[ I''(\psi) = I'(\psi) \] \( \ldots (8) \)

Step 3
Now consider \( g^{1} \). \( g^{1} \) is an \( S \)-bound function, and so
\[ I^n(\psi) = I'(\psi) = I''(\psi) \quad \text{by (8)} \]
i.e. \[ I^n(\psi) = I'(\psi) \]

Further

\[ I^n_\beta(T') = I^n_\beta(T') \land \beta(T) \quad \text{by lemma D.2} \]
\[ = \phi^n_\beta(T') \land \{ \delta^n(T') \land \beta(T) \} \land \beta(T) \]
\[ = \phi^n_\beta(T') \land \{ \delta^n(T') \land \beta(T) \} \land \beta(T) \]
\[ = \{ \phi^n_\beta(T') \land \beta(T) \} \land \beta(T) \]
\[ = \{ \phi^n_\beta(T') \land \beta(T) \} \land \beta(T) \]
\[ = I^n_\beta(T') \land \beta(T) \]
i.e. \[ I^n_\beta(T') = I^n_\beta(T') \quad \ldots (10) \]

But \( \beta^n \) and \( \beta^n \) are identical outside \( T' \), and so \( (10) \) implies that

\[ I^n_\beta(\psi) = I^n_\beta(\psi) \]

and then \( (9) \) implies

\[ I^n_\beta(\psi) = I^n_\beta(\psi) \quad \ldots (11) \]

Step 4

\[ I^n_\beta(T) = \phi^n_\beta(T) \land \beta(T) = \phi^n_\beta(T) \land \beta(T) = I^n_\beta(T) \land \beta(T), \]

and so

\[ I^n_\beta(T) \leq I^n_\beta(T) \quad \ldots (12) \]

Similarly \( I^n_\beta(T') = I^n_\beta(T') \land \beta(T), \)

and so

\[ I^n_\beta(T') \leq I^n_\beta(T') \quad \text{which implies} \]

\[ I^n_\beta(T') \leq I^n_\beta(T') \quad \ldots (13) \]

Combining \( (12) \) and \( (13) \) gives
\[ I_{r0}(T) \leq I_{r0}(T) \leq I_{r0}^n(T) \] ...(14)

But by lemma D.3, (14) implies

\[ I_{r0}^n(\psi) \leq I_{r0}^n(\psi) \leq I_{r0}^n(\psi) \]

where \( \leq \) if \( T \) is positive in \( \psi \), and \( \geq \) otherwise. But by (5) and (11) both of the outermost terms in this relationship are equal to \( I'(\psi) \), and so

\[ I'(\psi) \leq I_{r0}^n(\psi) \leq I'(\psi) \]

i.e. \( I_{r0}^n(\psi) = I'(\psi) \)

as required. QED

Theorem 7.1

Let \( \psi \) be a tree with a totally consistent bounding function \( \beta \).
If \( T \) is any positive subtree of \( \psi \) such that
\[ I(\psi) \wedge B(T) = \emptyset, \]
then \( B[T\#0] \) is another totally consistent bounding function on \( \psi \).

Proof

I will show that \( I_{\beta'[T\#0]}(\psi) = I(\psi) \) for any set \( X \) and any \( \beta' \models \beta \).

Assume without loss of generality that \( \beta'(T) = B(T) \). Then

\[ \emptyset = \beta'(T) \wedge I(\psi) \] (given)

\[ = \beta'(T) \wedge \{ I_{\beta'[T\#0]}(\psi) / B'(T) \} \cup \{ I_{\beta'[T\#0]}(\psi) \wedge B'(T) \} \]

by lemma D.4

\[ = I_{\beta'[T\#0]}(\psi) \wedge B'(T). \]

So

\[ I(\psi) = I_{\beta'(\psi)} \]

as \( \beta' \) is totally consistent

\[ = \{ I_{\beta'[T\#0]}(\psi) / B'(T) \} \cup \{ I_{\beta'[T\#0]}(\psi) \wedge B'(T) \} \]
Thus
\[ I(\psi) \leq I_{\beta'}[T\#\emptyset](\psi) \quad \ldots (1) \]

Now \( I_{\beta'}[T\#\emptyset](T) \leq I_{\beta'}[T\#X](T) \leq I_{\beta'}[T\#\emptyset](T) \), and as \( T \) is positive in \( \psi \) and using lemma D.3
\[ I_{\beta'}[T\#\emptyset](\psi) \leq I_{\beta'}[T\#X](\psi) \leq I_{\beta'}[T\#\emptyset](\psi) \quad \ldots (2) \]

Now \( \beta'[T\#\emptyset] \perp \beta \) and so is a consistent bound, and it follows that
\[ I_{\beta'}[T\#\emptyset](\psi) = I(\psi) \quad \ldots (3) \]

Combining \( (1) \), \( (2) \) and \( (3) \) gives
\[ I(\psi) \leq I_{\beta'}[T\#X](\psi) \leq I(\psi) \]

as required. QED

**Theorem 7.3**

If \( \{T_1, T_2, \ldots, T_m\} \) is a covering set for a tree \( T \), so is any superset formed by adding positive disjoint subtrees.

**Proof**

Let \( \{T_1, \ldots, T_m, T_{m+1}, \ldots, T_n\} \) be such a superset. Then
\[
I( \{T[T_1 \perp \perp; \ldots; T_{m+1} \perp \perp; \ldots; T_n \perp \perp] \} ) \\
= I(\{T[T_1 \perp \perp; \ldots; T_{m} \perp \perp][T_{m+1} \perp \perp; \ldots; T_n \perp \perp]\} ) \\
\leq I(\{T[T_1 \perp \perp; \ldots; T_{m} \perp \perp]\} ) \\
= 0
\]

by lemma D.3, and as each of \( T_{m+1}, \ldots, T_n \) is positive in \( T \). QED
Theorem 7.4

The set of the interpretations of all zero-order primitives of a tree T is a covering set for T.

Proof

Structural induction on the composition of T. If T is a primitive then the result is trivial. Otherwise let \( T = T_1 \langle \text{op} \rangle T_2 \) for some set operation \( \langle \text{op} \rangle \), and let \( P_1^* \) and \( P_2^* \) be the sets of positive primitives from \( T_1 \) and \( T_2 \) respectively. Then if \( \langle \text{op} \rangle \) is \( \cup \) or \( \cap \) the set of positive primitives of T is \( P_1^* \cup P_2^* \), and setting all these to \( \bot \) results in a null set as the total interpretation. If \( \langle \text{op} \rangle \) is \( \setminus \) then \( P_1^* \) is the set of positive primitives of T, and the result also holds. \( \Box \)
This appendix is concerned with a few results required for chapter 8, the chapter on the statistical analysis of algorithms. In this appendix $\phi(x)$ refers to the cumulative normal distribution function,

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{v^2}{2}} dv$$

**Lemma E.1**

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{u+ov} e^{-\frac{v^2}{2}} dv = e^{\mu+\frac{1}{2}o^2} \phi(x-o)$$

**Proof**

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{u+ov} e^{-\frac{v^2}{2}} dv = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{u+\frac{1}{2}o^2} e^{-\frac{1}{2}o^2+ov-\frac{v^2}{2}} dv$$

$$= \frac{1}{\sqrt{2\pi}} e^{u+\frac{1}{2}o^2} \int_{-\infty}^{x} e^{-\frac{1}{2}(v-o)^2} dv$$

$$= \frac{1}{\sqrt{2\pi}} e^{u+\frac{1}{2}o^2} \int_{-\infty}^{w} e^{-\frac{1}{2}w^2} dw$$

$$= e^{u+\frac{1}{2}o^2} \phi(x-o)$$

QED

**Lemma E.2**

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} (e^{u+ov})^2 e^{-\frac{v^2}{2}} dv = (e^{u+o^2})^2$$

**Proof**
Lemma E.3

For $x<0$,

$$\phi(x) < -\frac{1}{x/2\pi} e^{-\frac{1}{2}x^2}$$

Proof

Given $x$, consider the pair of functions

\[ f(y) = e^{\frac{1}{2}y^2 - xy} \quad \text{and} \quad g(y) = e^{-\frac{1}{2}y^2}. \]

Then $f(y)/g(y) = e^{\frac{1}{2}(x-y)^2}$ and so $f(y) > g(y) > 0$ for all $y$. Thus

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} f(y) \, dy > \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} g(y) \, dy = \phi(x)$$

and the integral on the left hand side is equal to
\[
\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{1}{2}x^2 - xy} dy = \frac{1}{\sqrt{2\pi}} \left[ -\frac{1}{x} e^{\frac{1}{2}x^2 - xy} \right]_{y=x}^{y=\infty} \\
= -\frac{1}{x\sqrt{2\pi}} e^{-\frac{1}{2}x^2}
\]

as required. QED
This appendix contains the vector expressions required for the calculation of the minimum distance function in robmod; that is, vector expressions for the candidate minimum distances between entities.

Case 1: Vertex to Vertex

Let the vertices be given by vectors \( u \) and \( v \) (as in figure F.1a). Then the distance between them, \( D \), is given by

\[
D^2 = (v-u)^2
\]

(I use the convention \( x^2 = x.x \), where "\( . \)" denotes the scalar product operator.)

Case 2: Vertex to Edge

Let the vertex be given by a vector \( p \), and let the edge be delimited by vectors \( u \) and \( v \). Let

\[
e = v-u
\]
and calculate

$$\lambda = \frac{(p-u) \cdot e}{e^2}$$

(figure F.1b). Then the minimum distance exists to an interior point of the edge only if $0 < \lambda < 1$, and then the distance, $D$, is given by

$$D^2 = (u - p + \lambda e)^2$$

**Case 3: Edge to Edge**

Let the edges be delimited by the vector pairs $(u_1, v_1)$ and $(u_2, v_2)$, and define

$$e_i = v_i - u_i \quad \text{for } i = 1, 2$$

(see figure F.2). Let

$$\lambda_i = \frac{(e_i \cdot (u_i - u_j))e_j^2 - (e_j \cdot (u_i - u_j))(e_i \cdot e_j)}{e_i^2 e_j^2 - (e_i \cdot e_j)^2}$$

for $i = 1, 2$ and $j = 3-i$. Then the minimum distance exists to the interiors of these lines only if $\lambda_1$ and $\lambda_2$ both exist and both lie in $(0, 1)$, and then the distance, $D$, is given by

$$D^2 = (\lambda_1 e_1 - \lambda_2 e_2 + u_1 - u_2)^2$$

**Case 4: Vertex to Face**

Let the face be given by the equation

$$n \cdot x + d = 0,$$

where $n$ is a unit vector, and let the vertex be given by $p$ (figure F.3). Then the minimum distance between these entities, $D$, is given by

$$D^2 = (n \cdot p + d)^2$$
...and it exists only if the foot of the perpendicular,

\[ p - (n \cdot p + d)n \]

exists on the shape (as determined, in robmod, by ray-casting).

\[ n \cdot x + d = 0 \]
Appendix G: PMC in Four Dimensions

This appendix is a discussion of a particular solution to the point-membership classification (PMC) function, as required for the robmod implementation of the four-dimensional version of baseNOD(). The first section of the appendix contains a formal statement of the problem and a restatement in terms of a problem that is more easily visualised, and the second section contains three solutions to this problem.

G.1 The Problem

The need to solve PMC occurs as part of the four-dimensional version of baseNOD(), when we want to determine whether or not a line segment, which has been formed by the intersection of a number of half-spaces, lies inside or on the clash hypervolume. I shall decorate four-component vectors with a caret (e.g. \( \hat{R} \)), and sometimes write four-component vectors in the form \( (x, \mu) \), where \( x \) is a three-component vector.

The line segments that I consider in baseNOD() are formed by intersecting a line with every other hyperplane in the localised CSG representation, and so we can always choose a point, on the interior of each line segment, which does not lie on any hyperplanes other than those in which the entire line lies. Further, to simplify this discussion we can choose the origin of the coordinate system to be such a point, and in such a coordinate system each hyperplane is given by an equation of the form

\[ \hat{n} \cdot \hat{x} = 0 \]

where \( \hat{n} \) is a normal to the hyperplane, and the line is given by an equation of the form

\[ \hat{x} = \lambda \hat{t} \]

for some parameter \( \lambda \).
The hyperplanes are the boundaries of the linear half-spaces which are bound to the leaves in a CSG tree. This CSG tree represents a Boolean function, which can be written in the form

\[ F(b_1, b_2, \ldots, b_m) \]

for Boolean primitives \( b_1, \ldots, b_m \).† and when I wish to consider this function to be a function of the half-space Boolean functions \( f_1(x), \ldots, f_m(x) \) I may write \( F \) as

\[ F[f_1, f_2, \ldots, f_m](x) \]

to emphasise that the dependency on \( x \) is common. In the case we are considering each \( f_i \) is of the form

\[ f_i(x) = (n \leq 0) \] .

Further we can write this underlying Boolean function in such a way that the set, which is represented by this function, is not null iff there exists a point \( x \) such that

\[ F[f_1, f_2, \ldots, f_m](x) \] is true.

In contrast to my discussion of tree substitutions (Chapter 6), where I was at pains to point out that each leaf corresponded to a different primitive, in this discussion it is helpful to consider each of the half-space functions to be distinct. We can do this without loss of generality as we can always identify any colinear normals. For example, if we have in two-dimensions a CSG representation of a shape that is given by \((AA(BUC))\), where \( A \) is the half-space \( x \leq 1 \), \( B \) is the half-space \( y \leq 1 \), and \( C \) is the half-space \( x \geq 1 \), then the Boolean function \( F \) could be considered as a function of two predicates, namely \( a \) and \( (b \ or \ not \ a) \), where \( a \) iff \( x \leq 1 \), and \( b \) iff \( y \leq 1 \).

With this notation we can specify the PMC problem that we need to solve as follows:

† Such functions are treated more formally in Appendix A.
Problem PMC1

Given a number of distinct linear half-spaces, as given by the functions

\[ f_i(x) \equiv ( n_i \cdot x \leq 0 ) \]

and that contain the line \( x = \lambda l \) in their surfaces, and given a Boolean function \( F \) which takes these functions as arguments, say whether or not there exists an interior point \( x \) such that

\[ F[f_1, f_2, \ldots, f_m](x) \text{ is true.} \]

This problem has a dual, as given by:

Problem PMC2

Given a number of distinct linear half-spaces, as given by the functions

\[ g_i(x) \equiv ( n_i \cdot x \leq 0 ), \]

and given a Boolean function \( F \) which takes these functions as arguments, say whether or not there exists an interior point \( x \) such that

\[ F[g_1, g_2, \ldots, g_m](x) \text{ is true.} \]

To show that these problems are duals, we have to show that an algorithm to solve either problem can be used to solve the other. So, firstly, assume that we have an algorithm to solve PMC1. Then given an instance of problem PMC2 create the instance of problem PMC1 with

\[ I = (0,0,0,1), \text{ and } \bar{n}_i = (n_i,0). \]

Then

\[ F[f_1, \ldots, f_m](x, \alpha) \text{ iff } F[g_1, \ldots, g_m](x), \]
and so the answer to these two problems is the same. Conversely, if we have an algorithm to solve PMC2 and are given an instance of problem PMC1 then we can choose a coordinate system in which \( \mathbf{1} = (0,0,0,1) \) and then each \( \mathbf{n}_1 \) must be writable in the form \( (n_1,0) \) for some \( n_1 \). Then the answer to the two problems is the same.

This result means that we can always rewrite our original PMC problem, namely PMC at an interior point of a line in four-dimensions, into another PMC problem, namely PMC at a vertex in three dimensions. This duality should not surprise us unduly, as a standard method of solving PMC at an interior point of a line in three dimensions is to consider a slice of space normal to the line, and so in effect transform the problem into one of PMC at a vertex in two dimensions. However, this duality is especially useful in our three-dimensional case as we generally have little intuition for working in four-dimensions.

\subsection*{G.2 Solving Problem PMC2}

In the last section I have shown how the PMC problem, that has to be solved in order to implement the four-dimensional version of \textsc{baseNOD(\ldots)}, can be recast in terms of a PMC problem in three-dimensions. In this section I will consider the solution to problem PMC2 in three cases.

\section*{Case 1: The Planes Meet in a Line}

For problem PMC1 this case corresponds to where the hyperplanes meet in a two-dimensional surface. The solution to this problem is reasonably well known (for PMC2) as it is often used in three-dimensional geometric modelling systems. The idea is that one takes a "section" perpendicular to the line and so transforms the problem into one of PMC at a vertex in two-dimensions, and this in turn is solved by walking around the vertex, and considering the one-dimensional boundaries of half-spaces that one intersects in doing so.
Case 2: Three Planes Meet at a Point

Note that this case corresponds to there being three linearly independent half-space functions in the localised CSG representation, and that any other case where there are three or fewer half-space functions in the localised representation is covered by case 1. I have found in practice that cases 1 and 2 are the only ones that rob-mod encounters. The solution to case 2 is, in fact, simpler than the solution to case 1, and is given by the following theorem:

Theorem G.1

Consider problem PMC2. If F is the Boolean function underlying a CSG representation of a shape and takes as arguments three linearly independent half-space functions \( g_1, g_2, g_3 \), then the set represented by this function is null iff \( F \) is a contradiction (i.e. if \( F \) always returns "false").

Proof

The set represented by the localised tree is not null iff we can find an \( x \) such that

\[
F(g_1, g_2, g_3)(x) \text{ is true.}
\]

If we can find such an \( x \) then \( F(g_1(x), g_2(x), g_3(x)) \) is true, and so \( F \) is not a contradiction. Conversely, if \( F \) is not a contradiction then we can find boolean constants \( b_1, b_2, b_3 \) such that \( F(b_1, b_2, b_3) \) is true, and so any point \( x \) that satisfies

\[
\begin{align*}
\hat{n}_i \cdot x &< 0 \quad \text{for } i=1,2,3 \\
\hat{n}_i &\begin{cases} 
\hat{n}_i & \text{if } b_i \\
-\hat{n}_i & \text{otherwise}
\end{cases}
\end{align*}
\]

\( b_i \)
will satisfy \( F[\vec{s}_1, \vec{s}_2, \vec{s}_3](x) \). But as the \( \vec{n}_i \)'s are linearly independent we can always find such an \( x \), and so the set represented by the localised tree is not null.

Thus in this case we never have to look for a point that is inside a set, and we can test \( F \) for being a contradiction simply by calculating \( F \) for all eight possible combinations of boolean inputs.

**Case 3: Anything Else**

This case covers anything where more than three distinct planes meet in a point. The contradiction method of case 2 will not always work here as the \( \vec{n}_i \)'s are not linearly independent, although if the underlying Boolean function is a contradiction it is certainly true that the set it represents is null. To solve this problem in general we could use the following generate-and-classify strategy:

1) For each pair of distinct planes, intersect them to form two infinite candidate edges, one each side of the origin.

2) Apply PMC at an interior point of each candidate edge.

As in case 1, each PMC problem here is concerned with an interior point of a line, and so can be solved. The main disadvantage with this solution is its potential time complexity; I have not yet implemented this method for the function `clash` in robmod.
References

[Aho et al 74]

[Ahuja et al 80]
"Interference Detection and Collision Avoidance Amongst Three Dimensional Objects" by N Ahuja, RT Chien, R Yen & N Bridwell, proc. 1st AAAI, Stanford, August '80.

[Ambler et al 74]
"Inferring the Positions of Bodies from Specified Spatial Relations" by A.P. Ambler, R.J. Popplestone & I.M. Bellos, Artificial Intelligence 6, No.2, pp157-174.

[Baer et al 79]

[Balila 82]
"A Study of Computer-Controlled Manipulators" by Mazin A. Balila, Geometric Modelling Project, University of Leeds (Internal Report).

[Balila & de Pennington 83]

[Baumgart 74]
"Geometric Modelling for Computer Vision" by Bruce G. Baumgart, October '74 Ph.D Thesis, reprinted as University of Stanford A.I. Memo, AIM-249.

[Boyse 79]
"Interference Detection Among Solids and Surfaces" by John W. Boyse, Comm. ACM, Volume 22 number 1, January '79.

[Boyse & Rosen 80]
"GMSOLID - A System for Interactive Design and Analysis of Solids" by John W. Boyse & Jeffrey M. Rosen, General Motors Research Laboratories, GMR-3451, October '80.

[Braid 73]

[Braid et al 78]
"Stepwise Construction of Polyhedra in Geometric Modelling" by I.C. Braid, R.C. Hillyard, I.A. Stroud, CAD Group Document number 100, Computer-Aided Design Group, University of Cambridge Computing Laboratory.
[Braid 79]
"Geometric Modelling — Ten Years On" by I.C. Braid, CAD Group Document number 103, Computer-Aided Design Group, University of Cambridge Computing Laboratory.

[Brooks 82]
"Solving the Find-Path Problem by Representing Free Space as Generalised Cones" by Rodney A. Brooks, MIT AI Memo No.674, May '82.

[Brooks 83]
"Planning Collision Free Motions for Pick and Place Operations" by Rodney Brooks, AI Lab Memo 725, May '83, MIT, Boston, Ma.

[Brooks & Lozano-Perez 82]
"A Subdivision Algorithm in Configuration Space for Findpath With Rotation" by R A Brooks & T Lozano-Perez, AI Lab Memo No 684, December '82, MIT, Boston; also in proc. IJCAI '83.

[Brown 81]
"Some Mathematical and Representational Aspects of Solid Modelling" by Chris Brown, IEEE PAM-I Vol 3 No 4, July '81.

[Brown 82]

[Burstall 69]

[Cameron 81]
"BRIDGET — A Geometric Body Modeller" by Stephen Cameron, Department of Artificial Intelligence Working Paper No.96, September '81.

[Cameron 82]
"The Clash Detection Problem" by Stephen Cameron, Department of Artificial Intelligence WP126, September '82.

[Coons 74]

[Dieudonné 60]

[Dooner 81]
"Computer Simulation to Aid Robot Selection" by Mike Dooner, presented to SERC Vacation School in Robot Technology, University of Hull, September '81.

[Featherstone 82]
"A program for simulating Robot Dynamics" by Roy Featherstone, DAI WP116, Department of Artificial Intelligence, University of Edinburgh, August '82.

[Feller 68]
"An Introduction to Probability Theory and its Applications — Volume 1" by W. Feller, 3rd Ed. 1968, Wiley N.Y.
[Foley & Van Dam 82]
"Fundamentals of Interactive Computer Graphics" by J.D. Foley and A. Van Dam, Addison-Wesley 1982.

[Franklin & Barr 81]
"Faster Calculation of Superquadric Shapes" by WR Franklin and AH Barr, IEEE Comp. Graphics & Applications Vol1 No3, July '81.

[Green 81]
"Clash Detection Algorithm" by John Green, internal report, University of Nottingham, Dept. Production Engineering, 1981.

[Halmos 63]
"Lectures on Boolean Algebras" by Paul Halmos, Van Noostrand, 1963.

[Hironaka 75]

[Hunter & Steiglitz 79]
"Operations on Images Using Quad Trees" by G.M. Hunter and K. Steiglitz, Pattern Analysis and Machine Intelligence, Volume 1 number 2.

[Johnson 78]

[Kernighan & Ritchie 78]

[Kingsley et al 81]

[Lozano-Perez 80]
"Spatial Planning: A Configuration Space Approach" by T Lozano-Perez, MIT AI Memo No 605, December '80.

[Lozano-Perez 83]

[Lozano-Perez & Wesley 79]

[Mendelson 75]
"Introduction to Topology" by Bert Mendelson, 3rd Ed, Allyn & Bacon, Boston 1975.

[Meyer 81]

"RAPT: A Language for Describing Assemblies" by R.J. Popplestone & A.P. Ambler, Industrial Robot September '78, pp.131-137 (Also as DAI Research Report No.79).

"A Spatially-Segmented Solids Database - Justification and Design" by K.M. Quinlan & J.R. Woodwark, School of Engineering, University of Bath - proc. CAD 82.


"Geometric Modelling of Mechanical Parts and Machining Processes" by A.A.G. Requicha and H.B. Voelcker, COMPCONTROL '79, Sopron, Hungary.


"Ray Casting for Modelling Solids" by SD Roth, Comp Graphics & Image Processing 18, pp.109-144, February '82.


"Exploiting Spatial and Structural Locality in Geometric Modelling" by Robert B. Tilove, Ph.D Thesis, College of Engineering and Applied Science, University of Rochester, October 81. Also as TM 38, Production Automation Project.


"Geometric Modelling of Mechanical Parts and Processes" by Herbert B. Voelcker & Aristides A. G. Requicha, Computing magazine, December '77.

"A Hidden Surface Algorithm for Computer Generated Halftone Pictures" by John E. Warnock, TR 4-15, Department of Computer Science, University of Utah, 1969.


"Polygon Decomposition and Alternating Sum" by Tony Woo, Department Industrial and Operations Engineering, University of Michigan, TR82-7, 1982.

"The Derivation of Graphics from Volume Models by Recursive Division of the Object Space" by J.R.Woodwark & K.M.Quinlan, School of Engineering, University of Bath, proc. CAD 80.


"Graphical Input to a Boolean Solid Modeller" by J.R.Woodwark & A.F.Wallis, School of Engineering, University of Bath, proc. CAD 82.
### Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \beta )</td>
<td>81, 6.3</td>
</tr>
<tr>
<td>( O(N), \Omega(), \Theta() )</td>
<td>53, 4.1</td>
</tr>
<tr>
<td>([, ])</td>
<td>82, 6.3</td>
</tr>
<tr>
<td>analyse</td>
<td>130, 8.7</td>
</tr>
<tr>
<td>baseNOD()</td>
<td>93, 7.1</td>
</tr>
<tr>
<td>Bath algorithms</td>
<td>54, 4.2</td>
</tr>
<tr>
<td>Bath_draw()</td>
<td>60, 4.2</td>
</tr>
<tr>
<td>Bath_NOD()</td>
<td>94, 7.2</td>
</tr>
<tr>
<td>Bernoulli trial</td>
<td>120, 8.3</td>
</tr>
<tr>
<td>Boolean algebra</td>
<td>16, 1.4.2</td>
</tr>
<tr>
<td>boundary</td>
<td>22, 2.1</td>
</tr>
<tr>
<td>boundary models</td>
<td>14, 1.4.1</td>
</tr>
<tr>
<td>bounded interpretation</td>
<td>81, 6.3</td>
</tr>
<tr>
<td>bounded tree</td>
<td>80, 6.3</td>
</tr>
<tr>
<td>bounding function, ( \beta )</td>
<td>80, 6.3</td>
</tr>
<tr>
<td>bubble test</td>
<td>50, 3.9</td>
</tr>
<tr>
<td>bubble-sort</td>
<td>52, 4.1</td>
</tr>
<tr>
<td>clash detection</td>
<td>40, 3.3</td>
</tr>
<tr>
<td>clash detection systems</td>
<td>172, 11.5</td>
</tr>
<tr>
<td>clash function in robmod</td>
<td>151, 10.2</td>
</tr>
<tr>
<td>ClashAT()</td>
<td>133, 9.1</td>
</tr>
<tr>
<td>ClashDC()</td>
<td>138, 9.2</td>
</tr>
<tr>
<td>Clashlin(), Clashran()</td>
<td>147, 9.5</td>
</tr>
<tr>
<td>closed set</td>
<td>21, 2.1</td>
</tr>
<tr>
<td>closure</td>
<td>22, 2.1</td>
</tr>
<tr>
<td>compactness</td>
<td>22, 2.1</td>
</tr>
<tr>
<td>compclash()</td>
<td>170, 11.4</td>
</tr>
<tr>
<td>complexity bounds; ( O(), \Omega(), \Theta() ) notation</td>
<td>53, 4.1</td>
</tr>
<tr>
<td>complexity; time, space, worst-case, expected</td>
<td>54, 4.1</td>
</tr>
<tr>
<td>composition operators for boxes and bubbles</td>
<td>89, 6.5</td>
</tr>
<tr>
<td>computational complexity</td>
<td>51, 4.1</td>
</tr>
</tbody>
</table>
configuration space 43, 3.5
connectedness 22, 2.1
consistent bounding function 82, 6.3
constant-density assumption 117, 8.2
continuous function 185, A.8
cost function, Tilove's 117, 8.2
covering set 97, 7.4
coverset() 104, 7.6
CSG models 15, 1.4.2
CSG tree 78, 6.1

dimension reduction 115, 8.1
distribution theorem 66, 4.4
divide-and-conquer 29, 2.4
downSB() 58, 4.2
downward theorem 86, 6.3

EMULA 42, 3.5
expected complexity 54, 4.1
explicit sweeping 160, 10.6
extrusion operator 28, 2.4

findpath problem 42, 3.5
four-dimensional interference detection 45, 3.7

generate-and-classify 56, 4.2

half space 18, 1.4.4
heuristics 104, 7.6
hidden-line removal 195, B.8; 68, 4.5
homogeneity assumption 119, 8.3

implicit sweeping 160, 10.6
initSB() 85, 6.3
inside() 64, 4.3
interference detection 37, 3.1
interior 22, 2.1
interpretation, of primitives and trees 79, 6.2

LARS 161, 10.7
line-membership classification (LMC) 56, 4.2
localisation 66, 4.4
location function 24, 2.3
loose bounds 107, 7.7

maxspeed() 133, 9.1
meta-level reasoning 48, 3.9
metric space 21, 2.1
mindist() 144, 9.4
minimum distance algorithms 138, 9.2
multiple interference detection 43, 3.6

NOD_bound() 102, 7.5
NONAME 161, 10.7
null object detection 40, 3.2; 93, 7.0

open mapping 185, A.8
open set 21, 2.1
order & sign of a subtree 96, 7.3

parameterised surface models 18, 1.4.3
partial ordering on functions, \( \preceq \) and \( \succeq \) 82, 6.3
pidgin-Algol 52, 4.1
plantSB() 86, 6.3
PMC 19, 1.4.5; 56, 4.2
projection 163, 10.9

quad-tree 197, B.8

r-sets 23, 2.2
ray-casting 61, 4.3
<table>
<thead>
<tr>
<th>Term</th>
<th>Page, Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ray_cast</td>
<td>63, 4.3</td>
</tr>
<tr>
<td>regular set</td>
<td>23, 2.2</td>
</tr>
<tr>
<td>regularisation</td>
<td>22, 2.1</td>
</tr>
<tr>
<td>rewrite rules</td>
<td>58, 4.2</td>
</tr>
<tr>
<td>rigid-body transformation</td>
<td>24, 2.3</td>
</tr>
<tr>
<td>robmod, ROBMOD</td>
<td>69, 5.0</td>
</tr>
<tr>
<td>S-bound [function]</td>
<td>82, 6.3</td>
</tr>
<tr>
<td>semi-algebraic set, semi-analytic set</td>
<td>30, 2.5</td>
</tr>
<tr>
<td>set operations</td>
<td>16, 1.4.2</td>
</tr>
<tr>
<td>set-membership classification</td>
<td>56, 4.2</td>
</tr>
<tr>
<td>sorting; lower bound for</td>
<td>53, 4.1</td>
</tr>
<tr>
<td>space complexity</td>
<td>54, 4.1</td>
</tr>
<tr>
<td>squares world</td>
<td>115, 8.1</td>
</tr>
<tr>
<td>structural induction</td>
<td>211, D.8</td>
</tr>
<tr>
<td>substitutions (on trees)</td>
<td>78, 6.1</td>
</tr>
<tr>
<td>sweeping methods</td>
<td>46, 3.8</td>
</tr>
<tr>
<td>S_NODE()</td>
<td>99, 7.4</td>
</tr>
<tr>
<td>teach mode of robot programming</td>
<td>41, 3.4</td>
</tr>
<tr>
<td>time complexity</td>
<td>54, 4.1</td>
</tr>
<tr>
<td>topological space</td>
<td>21, 2.1</td>
</tr>
<tr>
<td>totally consistent bounding function</td>
<td>82, 6.3</td>
</tr>
<tr>
<td>triangulable set</td>
<td>23, 2.2</td>
</tr>
<tr>
<td>upSB()</td>
<td>85, 6.3</td>
</tr>
<tr>
<td>upward theorem,</td>
<td>84, 6.3</td>
</tr>
<tr>
<td>Winged-Edge data-structure</td>
<td>15, 1.4.1</td>
</tr>
<tr>
<td>WIREFRAME</td>
<td>115, 8.1</td>
</tr>
<tr>
<td>worm</td>
<td>33, 2.6</td>
</tr>
<tr>
<td>worst-case complexity</td>
<td>54, 4.1</td>
</tr>
</tbody>
</table>