Convex hull generation, connected component labelling, and minimum distance calculation for set-theoretically defined models

CONVEX HULL GENERATION, CONNECTED COMPONENT LABELLING, AND MINIMUM DISTANCE CALCULATION FOR SET-THEORETICALLY DEFINED MODELS

Submitted by Dan Pidcock
for the degree of
Doctor of Philosophy
of the University of Bath
2000

Abstract

Three areas of geometric modelling have been studied in this thesis: the computation of the convex hull of a model, the labelling of the connected components of a model, and the calculation of the minimum distance between two objects in a model. The models considered for each area of research are solid models which have been defined set-theoretically, i.e. using computational solid geometry (CSG). Although the areas of research have been considered independently of the dimensionality of the models, methods developed in this work have been implemented for three-dimensional models.

The following new results are presented in this thesis:

- Computing the convex hull of a set-theoretically defined model. A point-set that represents the model is generated by one of three methods (one for polyhedral models, two for non-polyhedral models). The convex hull of this point-set is then found by using an existing convex hull algorithm.
- A heuristic method of computing the connectivity between two points within a given model. This works using a divide-and-conquer method to attempt to find a series of points which are connected.
- A connected-component labelling algorithm utilising a binary tree storage structure for a model by dividing a model and storing the divided model in a binary tree. Neighbouring nodes in the binary tree are examined to compute connected components.
- An algorithm for the computation of the minimum distance between two models by dividing the models recursively and concentrating the division on areas within which the closest points between the models are considered more likely to be found.

Acknowledgements

Thanks to Rachel for her great support, and to everyone else who has helped me keep going and made life fun throughout the duration of this work, especially Roger, Sylvia, Jo, Adrian, Kev, Dave, James, Jan, Jakob, Irina, and not forgetting Nougat and Caramel.

I would like to thank the Department of Mechanical Engineering, University of Bath for funding this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of geometric modelling is large and has many applications, for example in robotics and computer aided design. The majority of commercial geometric modelling applications, however, use the boundary representation method to store the geometric models, and rarely utilize set-theoretic modelling, largely because of the lack of effective solutions to key problems. Because set-theoretic modelling is a more intuitive and human-centred approach to modelling, several boundary representation modelling applications (for example ACIS) actually use set-theory for shape definition when interacting with the user. Set-theoretic modelling also generalizes to higher numbers of dimensions more easily than boundary representation modelling.

This thesis concentrates on set-theoretic modelling and some of the unsolved problems which currently make it less attractive for commercial application. The three areas which have been studied in detail are convex hulls, connected component labelling, and minimum distance. Each of these problems has already been the subject of research and, although several efficient algorithms have been found for other modelling representations such as boundary representation, little work exists for set-theoretically defined models. The new work presented in this thesis consist of the following:

- A method of computing the convex hulls of set-theoretically defined objects by generating points that are sent to an existing point-set algorithm.
- An overview of a multi-dimensional method for directly computing convex hulls.
- A heuristic method that computes the connectedness of two points in a model.
- A connected component labelling algorithm utilising a binary tree storage for a model.
- An algorithm to compute the minimum distance between two models.

An introduction to geometric modelling is provided in Chapter 2 of this thesis, including boundary and set-theoretic modelling representations and binary trees. A binary tree is a data structure which enables many operations to be performed efficiently on set-theoretic geometric models. The sVLIs set-theoretic kernel modeller is then outlined, followed by a description of the adaptive recursive spatial division problem-solving technique which has been used in my proposed solutions of the three problem areas.

A chapter has been devoted to each area of research. Within each chapter an introduction to and the definition of the problem being studied is given, followed by a literature survey of existing research. Methods of computing the properties have been proposed and results from my implementation of these methods are given. At the end of each of these three chapters there is a section detailing areas of interest for further research and conclusions from the work in that chapter. Finally, Chapter 6 summarizes the conclusions.

The research has been implemented using the sVLIs [1] [10] set-theoretic kernel modeller developed by Adrian Bowyer at the University of Bath.

[1]SVLIs is pronounced soo-liss.

Chapter 2

Background

2.1 Geometric modelling

The main geometric representation in commercial use is the *boundary representation*, where the vertices, edges and faces of a model are each stored together with topological information about their adjacency relationships. Set-theoretic modelling, also known as constructive solid geometry (CSG), is an alternative form of modelling, in which simple shapes are combined using set-theoretic operators to create more complex models. Set-theoretic modelling is a more intuitive way of modelling than boundary representation and, for this reason, most boundary representation modellers employ CSG front-ends for shape definition. Set-theoretic solid models can also be dealt with more consistently and the representation generalizes well to n-dimensions. These two forms of solid modelling have been outlined in sections 2.1.1 and 2.1.2, given below. More detailed descriptions can be found in the work of Woodwark [79] and Rooney [62].

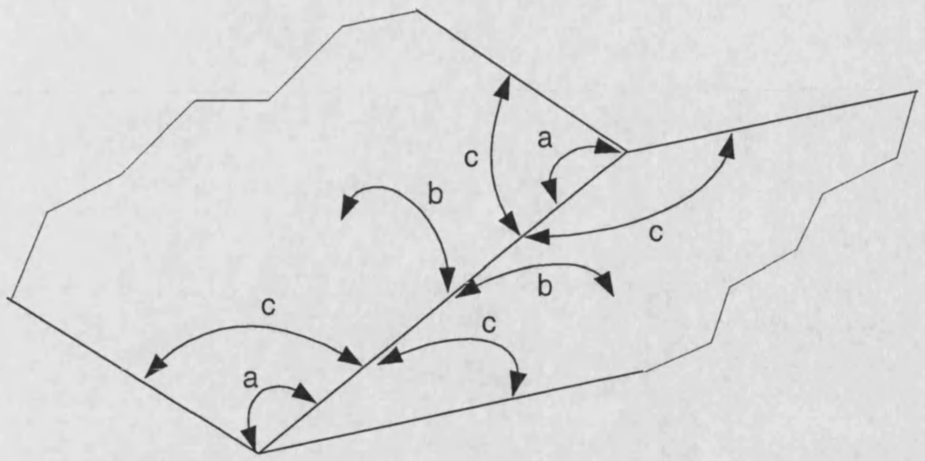


Figure 2.1: Baumgart's winged-edge data structure (after 7.1 by Woodwark [79]).

2.1.1 Boundary representation (B-rep)

A boundary representation model explicitly stores the faces, edges, and vertices of an object, along with the topological relationship between these elements. The most common data structure for this is Baumgart's winged-edge data structure. This structure provides links between every edge in the model and the two end vertices of the edge (a), between the edge and the two adjacent faces (b) and between the edge and the four other edges that share a face and a vertex with the edge (c), as illustrated in figure 2.1.

There is no intuitive concept of solid models when using boundary representation as it represents the shell of a model. Storing a solid model requires the model to be tagged to identify which side of the faces is solid.

An advantage that boundary representation models have over set-theoretic models is that the vertex, edge and face information is explicitly stored in b-rep models, therefore operations such as finding the closest two points between convex polyhedral models can be performed by searching the model's data structure to find the points. For more complicated shapes such as NURBS-based models, and generally for non-convex models, less efficient techniques must be used. Local minima can be found by using calculus, then global optimisation methods such as simulated annealing must be applied to find the closest points. These techniques are not specific to geometrical modelling and so are not further considered in this work. Connectivity information is explicitly stored within a model's data structure on creation; however finding the connectivity between two separate b-rep models that exist in the same space is not as straightforward and requires intersection testing of vertices and faces for the models.

2.1.2 Set-theoretic modelling

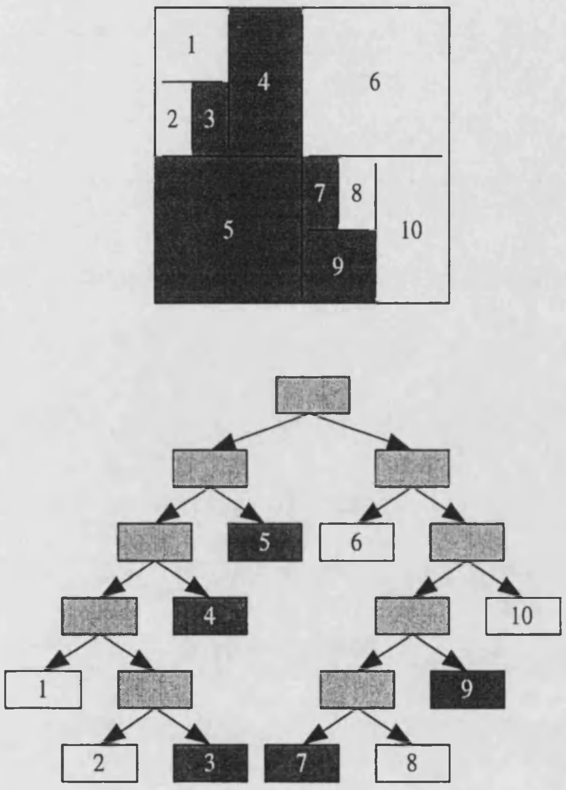
Defining an object using set-theoretic modelling is done by combining primitive objects using the following operators of set-theory: union, intersection and complement. The primitive objects chosen affect the types of solid models that can be defined; they can be entities such as cuboids, cylinders, and half-spaces, all of which can be defined as collections of polynomial inequalities. In solid modellers, the polynomial value of any point in space shows whether or not it is solid; if the value of a point is above a threshold value then that point is considered to be solid, conversely a point which has a value below the threshold is considered to be air. Points whose values are equal to the threshold value are considered to be on the surface of a model. Different set-theoretic solid modellers may use different quantities for the threshold value, and different conventions that define whether points with values above the threshold are solid, or points with values below the threshold are solid: for example in sVLIs the threshold value is zero, and points with values below the threshold are considered to be solid.

Set-theoretic models can be stored as a tree where the nodes are boolean operators and the leaves are the atomic primitive objects. In general, set-theoretic modellers need less storage space than boundary representation modellers, as only the equations of the half-spaces and the set-theoretic tree are stored - this is especially true for models in a space of high dimensionality. The disadvantage of storing model information in this non-evaluated way is that finding high-level geometrical information, such as the distance between two objects or counting the number of disjoint objects that exist in a model, requires significant computational effort.

The set-theoretic solid modeller in use at the University of Bath is the sVLIs set-theoretic kernel geometric modeller, which will be discussed in more detail in section 2.2 below.

2.1.3 2^n trees and Binary trees

Spatial data can be stored effectively in trees which break a rectilinear volume into smaller parts and group areas of similarity together. Two types of trees commonly used for storing such spatial data are 2^n trees and binary trees, an example of each is illustrated in figure 2.2. A 2^n tree divides $n-$dimensional space into 2^n parts at each level of the tree: for example two-dimensional space is divided into quadrants at each successive subdivision to create a quadtree [44], and similarly three-dimensional space is divided into eight cuboids at each subdivision to create an octree. Binary trees always divide the space into two parts at each tree level, regardless of the dimensionality of the space, and can be used to divide n-dimensional space by dividing each dimension in turn. Both types of tree can divide the volume at any position, but if they are restricted to divide the volume at the mid-point the split position need not be stored, requiring less memory and possibly making algorithms which use the trees more efficient at the cost of flexibility in the division of the volumes.



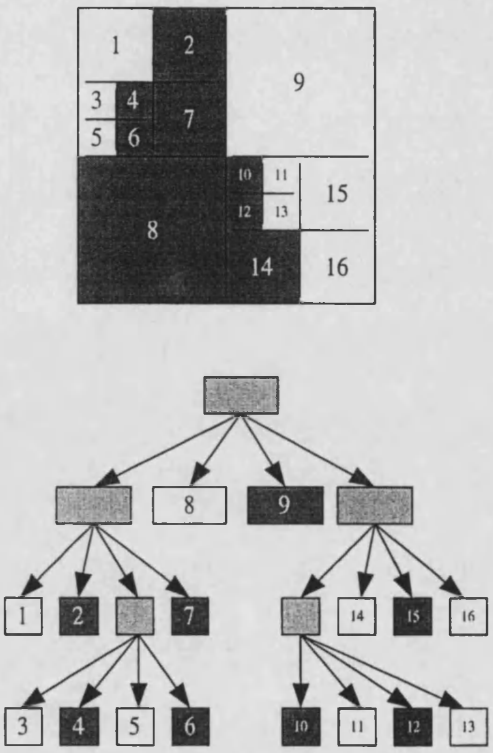


Figure 2.2: The binary tree and the 2^n tree required to store the same spatial data.

An advantage that binary trees have over 2^n trees is that extension to an arbitrary number of dimensions is more straightforward. For both types of tree the data stored at nodes must increase in dimensionality, but for 2^n trees the number of children that each non-leaf node has will also increase as n increases. At each non-leaf node in a binary tree, the node will still have two children, but the choice of dimensions that is being split by the children will increase: for a two-dimensional binary tree, each node is either split in the x- or y-axes to create two children, whereas in three-dimensions the node can be split in the x-, y- or z-axes to create two children. It is simpler to extend the data structure for a binary tree as the dimensionality increases; however it will also produce a deeper tree. In some cases the binary tree can be a more efficient way of storing a model (less nodes are required) as the data structure can match the shape of the model more closely.

2.2 The Svlis set-theoretic solid modeller

SVLIS is a set-theoretic geometric modeller, developed at the University of Bath by Adrian Bowyer [10], and is the modeller on which algorithms have been implemented for this research.

SVLIS uses the concepts outlined in section 2.1.2 (above). Points with negative values are considered to be within solid, points with positive values are outside the solid (i.e. they are in air), whilst points with zero value are on the surface of a model.

The simplest components used in sVLIS are primitives. There are currently six standard primitives in sVLIS : real numbers, half-planes, spheres, infinite cylinders, infinite cones, and tori. The modeller also allows any implicit function to be used as a primitive, thus giving access to many possible shapes. One type of primitive is real numbers, which enable arithmetic operations to be performed, while the remaining five primitives are half-spaces which divide space into two (possibly multiply-connected) regions, denoted in sVLIs as *solid* and *air*. The solid region contains all points that have a zero or negative value when substituted into the algebraic expression defining the primitive; all other points are in the air region.

Primitives may be combined using the four set-theoretic operators 'union' ($\bigcup$), 'intersection' ($\bigcap$), 'difference' ($-$) and 'symmetric difference' ($\triangle$) to create a set-theoretic tree with primitives at the leaves and operators at the nodes. In sVLIs such a tree is called a *set*. The four operators are reduced to $\bigcup$, $\bigcap$ and $-$ internally by sVLIs using standard Boolean re-write rules.

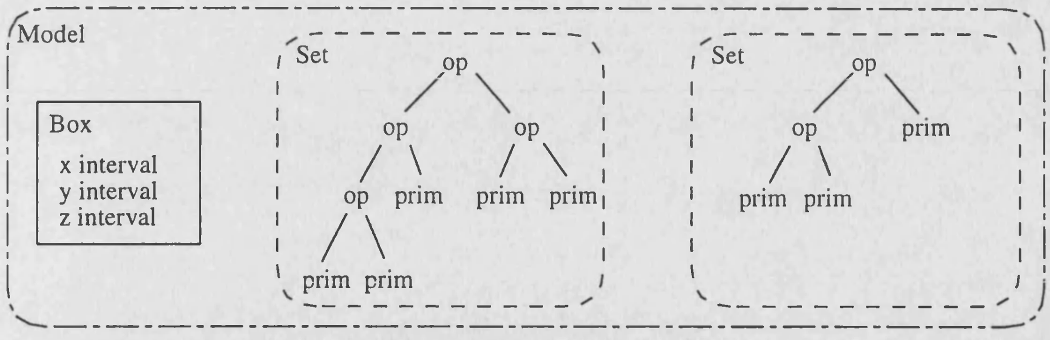
A *box* is a three-dimensional region of interest which is an axially-aligned cuboid; it is stored in sVLIs as three intervals[1], one for each dimension.

A *model* is the highest level entity in sVLIs and is defined as a box together with one or more sets. It defines a group of objects in the region of space that the box occupies.

The relationship between these elements of sVLIs is illustrated in figure 2.3.

The concepts of 'solid' and 'air' in sVLIs can be applied to different elements, namely primitives, sets, models and points. Primitives divide space into solid and air regions. When primitives are combined into a set, the set will have solid regions and air regions within it. If a model's box lies completely within the solid

[1]An interval is a continuous section of the real line between two values, and is written $[a, b]$ where a and b are the bottom and top end of the interval.



op = set–theoretic operator, prim = primitive

Figure 2.3: The relationship between the elements used in sVLIs .

region of the model's set, then that model is considered to be solid. The reverse is true for air models, and if a model's box contains solid and air regions of a set then it is considered to be a *surface model.* Given a point and a set, a *membership test* can be performed to test if that point is within the solid or air region of the set. For brevity, I usually refer to this classification of a point with a set as a point being (in) solid or air.

2.2.1 Pruning

If the surface of a primitive in the sets of a model does not pass through the model's box, then either the primitive must be completely solid within the box, or the box does not contain any part of the primitive, i.e. the primitive is air within the box. If the primitive is solid within the box, it can be replaced by the universal set U in the model's set for that box only. Conversely, if the primitive is air within the box, it can be replaced by the empty set $\emptyset$. The set for that box can now be simplified by using the basic set-theory rules $S \cup \emptyset = S$, $S \cap \emptyset = \emptyset$, $S \cup U = U$ and $S \cap U = S$.

This process reduces the complexity of the set-theoretic tree that is required to represent the set within a given box, and is known as *pruning a set to a box.*

2.2.2 Adaptive recursive spatial division

The box of a sVLIs model can be divided into two smaller *sub-boxes* by *cutting* it in any position within the box along one of the coordinate axes. The original model's set is pruned to each of the new sub-boxes to create two new *sub-models.* This *adaptive spatial division* [80] is applied recursively to the two sub-models. Terminating conditions for the recursion depend on the application of the divided structure; however a useful set of general-purpose conditions is to stop the recursion if the ratio of the volume of the sub-model's box to the volume of the original box is less than a given quantity, or if the sub-model contains less than a certain number of primitives. When dividing a model's box into two sub-boxes, the cut can be made in different places for each of the child boxes, so that the two child boxes overlap slightly. This is useful to prevent axially-aligned planar half spaces from being 'lost' if they coincide exactly with the cut position[2]. The *box swell factor* is the factor by which the child box is larger than the box would have been if the cut was made in the same position for each child box - a factor of zero will result in no overlap between the child boxes.

At the end of the division, three types of sub-models will exist:

- solid models whose box only contains solid;
- air models whose box only contains air;
- surface models whose box contains one or more primitives which cannot be further pruned and whose box may contain just air, just solid or a mixture

[2]This might, for example, be caused by floating-point rounding.

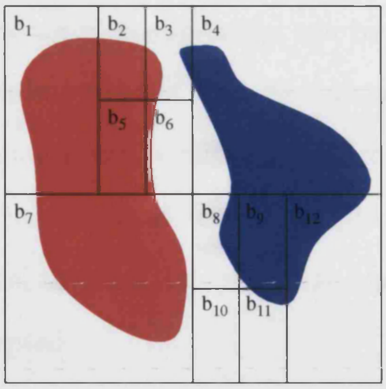


Figure 2.4: A two-dimensional divided model containing air, solid and surface sub-models.

of solid and air[3].

A divided model with sub-models of all three types is shown in figure 2.4. The sub-model with box b_5 is a solid model, the sub-model with box b_{10} is an air model and all of the other sub-models are surface models.

2.2.3 Effective solution of geometric problems by using adaptive recursive division

Adaptive recursive division can be a useful technique for solving a geometric problem by splitting a model into sub-models. The division must be halted at some point, which would commonly be defined either by the size of the sub-model's box (relative to the original model's box - referred to as the *division resolution*) or by the complexity of the sub-model (for example, the number of primitives in the sub-model). At this point the problem must be solved analytically. The chosen division resolution affects the complexity of the smallest sub-models for which the problem must be solved. If an intelligent analytical solution is used

[3]The way that sVLIs evaluates whether a box contains any surface uses interval arithmetic and is conservative so cannot reliably confirm that a box really does contain some surface.

then the complexity of the simplest sub-model can be set higher and the division resolution larger, so fewer sub-models need be considered and less division performed. The computational effort required for more intelligent solutions needs to be evaluated, and then compared to the computational effort and memory requirements of sub-model division, in order that the most efficient solution to any given problem is adopted.

Chapter 3

Convex hulls

3.1 Introduction

A set is convex if all line segments between any two points within the set lie entirely within the set. Given a set of objects S, the convex hull of S, $C(S)$, is the smallest convex set that contains S. Figure 3.1 shows example convex hulls of two-dimensional and three-dimensional discrete point sets $\{p_1, p_2, ..., p_8\}$ and $\{p_1, p_2, ..., p_7\}$ respectively.

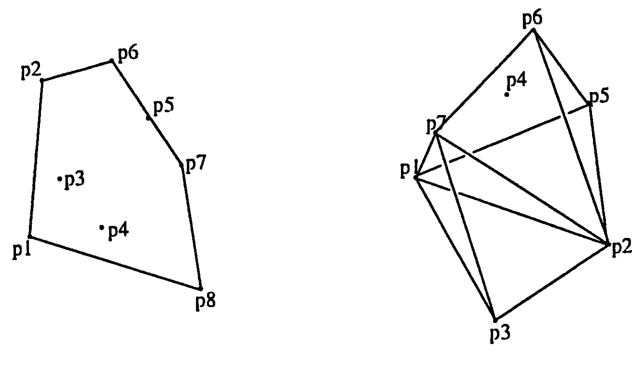


Figure 3.1: Example convex hulls of two- and three-dimensional point sets

The convex hull of a point set can be described by any of:

- the faces of $C(S)$;
- the vertex set, $Ver(S)$ which is the minimum subset of S such that $C(Ver(S)) = C(S)$;
- the set of hyperplanes, $H^*(S)$, by which $C(S)$ becomes the intersection of the half spaces bounded by $H^*(S)$.

In many applications of the convex hull an approximation of the hull is sufficient, provided that it can be found quickly. For example in a robot path planning application, instead of performing exact collision detection between the robots and obstacles, the approximate convex hulls of these objects can be used for initial tests and only objects whose approximate hulls intersect will then need to be examined for more accurate collision detection. It is essential that the approximate hulls found are guaranteed to be the same as or larger than the exact hull for this method to work.

Specific applications of convex hulls include:

- Collision detection and path planning. The convex hull can be used in path planning [52] (along with heuristics) to generate a non-optimal path to solve the travelling salesman problem.
- Machine vision, where the stable configurations that define the ways that a three-dimensional object may be positioned on a surface can be found from the convex hull. The convex hull can be further used if the image that the machine is recognising is silhouetted against a contrasting background, meaning that not all concave features on the object will be visible.

The convex hull of the object can be compared to the convex hull of the silhouette for preliminary image matching.

- Pattern recognition [63], by using the convexity of an image to classify it.
- Feature recognition, which uses convex decomposition, some implementations of which, for example Kim [42], require that the convex hull be found.
- Statistical problems in operational research [34].

3.2 Literature survey

The vast majority of work on convex hulls is on convex hulls of point sets, although other geometrical entities have been investigated such as sets of spheres [7]. Several point-set algorithms exist to find the two-dimensional convex hull [31, 34, 24, 83, 21] and the three-dimensional hull [27, 2], and general approaches have been proposed for the d-dimensional case [18, 39, 8]. Recently, research has become more focused upon numerically robust convex hull techniques [32] and convex hull algorithms for special hardware including parallel computers and multicomputers [22, 26, 83, 21], two-dimensional processor arrays with a reconfigurable bus system [55] and neural networks [20, 47]. In general the best efficiency that can be performed is $O(n \, log \, n)$: the closest that has been achieved in three-dimensions is that of $\Theta(n \, log \, n)$ by Preparata and Shamos [60].

3.2.1 Point set algorithms in d-dimensions

Methods of computing the convex hulls of point sets include *gift-wrapping,* developed by Chand and Kapur [18], and *beneath-beyond,* proposed by Kallay [39].

These methods are discussed in detail below. Borgwardt (1997) [8] presents an algorithm that combines gift-wrapping and a method of disregarding non-extremal points. Sugihara [72] revised the gift-wrapping algorithm in 1994 so that it is numerically robust. Preparata and Shamos [60] give the computational complexity of the gift-wrapping algorithm as $O(n^{[d/2]+1}) + O(n^{[d/2]} \log n)$ and the beneath-beyond technique as $O(n^{[d/2]+1})$

Gift-wrapping

Chand and Kapur (1970) [18] developed the *gift-wrapping* algorithm to find the convex hull of a d-dimensional point set $S = \{p_1, p_2, ..., p_n\}$. This starts with an initial facet of the convex hull and then rotates a plane about one of the edges of that face until the plane 'hits' one of the points in S. A new facet is created using the edge and the point found. The process is repeated until each edge of the convex hull has two adjacent facets.

The initial facet is found by generating a plane parallel to the axis of the first dimension and wrapping it around the point that has the smallest component in that dimension, in a similar way that the wrapping is performed for the main part of the algorithm.

The wrapping step is performed by using an existing facet F_0 and an edge e of that facet; a new facet F_1 is constructed from the edge e and a point p_k (that is not in F_0) which makes the angle between F_0 and F_1 as large as possible. Figure 3.2(a) illustrates this situation - p_5 would be chosen as the point from which to construct F_1. To find p_k, let n be the unit normal to F, let a be a unit vector normal to both the edge e and n and let v_k denote the vector p_2p_k, as illustrated in figure 3.2(b). In order to determine which point should be used,

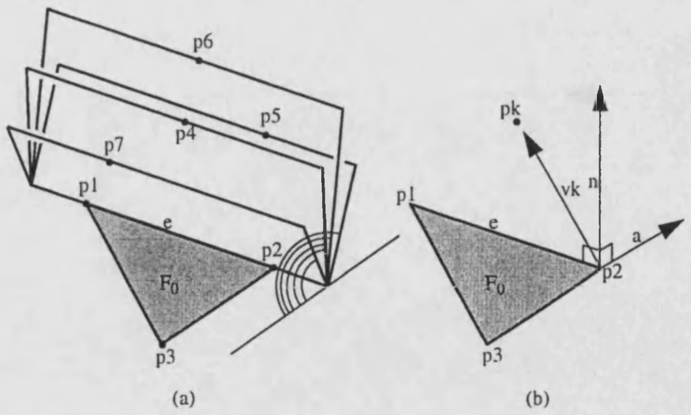


Figure 3.2: (a) The half-plane containing the facet formed by e and p_6 forms the largest angle with the half-plane containing F_0. (b) Values for calculating the angle.

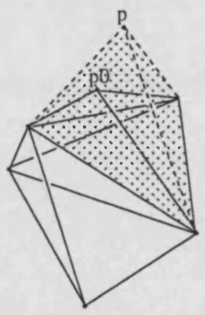


Figure 3.3: Extending the convex hull in the beneath-beyond on-line algorithm.

compute $\gamma_k = v_k.a/v_k.n$ for all of the points p_k not in F - the required point is that which maximises γ_k.

Beneath-beyond

The *beneath-beyond* method was proposed by Kallay in 1981 [39], again relating to finite point sets in d-dimensional space, but it has an advantage over the the gift-wrapping method of the *on-line* property. An on-line point-set convex hull algorithm is one which can be given one point at a time and which then computes the convex hull after it has been given each point.

The basic method starts with a given convex hull[1]. When a new point is introduced, test if the point is inside the convex hull: if it is then ignore it. Otherwise, add the point to the convex hull and remove all points from the convex hull that are in the cone beneath the new point. This is illustrated in figure 3.3, where p is the new point added to the convex hull and the cone is shown as the shaded area. In this case only one point p_o will be removed from the convex hull.

3.2.2 Point set algorithms in three dimensions

Finding the convex hull of a three-dimensional point set can be done more efficiently than the general d-dimensional case. Preparata and Shamos [60] suggest a technique in their book published in 1985 with optimal worst case time complexity of $\Theta(n \, log \, n)$ (i.e. *within* a constant multiple of $n \, log \, n$). Allison and Noga's [2] program *tetra* has expected running time of $O(n)$ (i.e. *less than* a constant multiple of n).

Preparata and Shamos's method

The algorithm works by using the 'divide and conquer' problem-solving technique. The points $\{p_1, \ldots, p_n\}$ are first sorted in any axis, and then the following recursive algorithm can be applied:

[1]The first $n+1$ non-coplanar points in n-dimensions can be used to create the initial convex hull.

convex_hull(S)

```
if number of elements in S < k then
    construct CH(S) by brute force
    return CH(S)
else
    $S_1 \leftarrow \{p_1, \dots, p_{\frac{n}{2}}\}$
    $S_2 \leftarrow \{p_{\frac{n}{2}+1}, \dots, p_n\}$
    $R_1 = convex\_hull(S_1)$
    $R_2 = convex\_hull(S_2)$
    $R = \text{merge}(R_1, R_2)$
    return($R$)
end
```

The merge step is key to the performance of the algorithm; it is performed by constructing a triangulation T around R_1 and R_2 using a gift-wrapping operation and removing points of R_1 and R_2 that are 'obscured' by T.

The first step in the construction of T is to create a facet of T. One way of doing this is to project R_1 and R_2 onto the coordinate plane perpendicular to the axis in which the points have been sorted and then to construct a common support line e' of the two projections. The line (planar edge) e' is the projection of an edge e of T. The plane through e and parallel to the axis in which the points have been sorted can be used as the starting facet of T.

The triangulation is now created by advancing around the two hulls. Let vertices of R_1 be denoted by a_i and vertices of R_2 by b_i. Let (a_2, b_1, a_1) be the last constructed facet of T. Vertices a' and b' must now be selected such that a' is

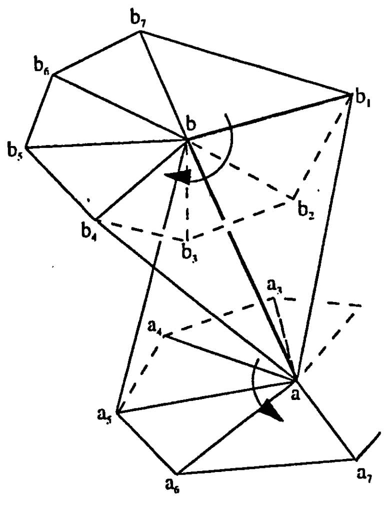


Figure 3.4: Illustration for Preparata and Shamos's 3-dimensional convex hull technique (after Preparata and Shamos's Fig 3.31)

connected to a_2 and b' is connected to b_2; the facet (a_2, b_1, a') forms the largest convex angle with (a_2, b_1, a_1) and the facet (a_2, b_1, b') forms the largest convex angle with (a_2, b_1, a_1). Of the two facets (a_2, b_1, a') and (a_2, b_1, b'), the one which forms the greatest convex angle with (a_2, b_1, a_1) is chosen, and the third vertex of that facet (a' or b') is added to T.

The removal of points in R_1 and R_2 that are 'obscured' by T are removed while finding the next facet. Referring to figure 3.4 for illustration, (b_1, b, a) is the last constructed facet of T, the vertices b_2 to b_7 are being considered for b' and a_4 to a_7 are being considered for a'. Let (b_s, b, a) be the facet that forms the largest convex angle with (b_1, b, a) (in the example s=4). Any edge (b, b_i) for $1 < i < s$ will be inside the final convex hull and can therefore be discarded from further consideration. Since the edges around vertex a have been partially scanned at an earlier stage in the process, the scan to find the vertex for a' should start at the last visited edge - (a, a_4) in the example.

Allison and Noga's method

Allison and Noga [2] have written a program *tetra* to compute the three-dimensional convex hull of a set of n points in (x, y, z) space which has expected running time of $O(n)$. The program uses a combination of divide and conquer [60] and incremental [40] approaches. Firstly, points that are definitely inside the convex hull are eliminated. Secondly an initial tetrahedron is found that consists of points definitely on the convex hull. The tetrahedron is then "grown" outwards with points from the point set until all possible points have been considered. A more detailed description of the program will now be given.

For the program, two sets of facets are used: w is for facets definitely on the convex hull and t is the set of candidate facets. The point set of which the convex hull is to be found is denoted by P.

The point elimination pre-processing step is performed by finding minimum and maximum points in each coordinate direction, and defining a prism using these points. All points within a cuboid which fits inside the prism are then deleted from P.

The initial tetrahedron is then constructed. The minimum and maximum points in each coordinate direction are found and a test is performed to ensure that they are not coplanar. If they are coplanar, then the program exits without having found the convex hull[2]. The minimum and maximum points in the x direction (x_{min} and x_{max}) are used to define a line segment. The point whose projection onto the x-y plane has the furthest perpendicular direction is found. This point, together with x_{min} and x_{max}, defines an initial facet f_1. Let $hp(f)$ denote the

[2]A simple (but less efficient) method of constructing an initial tetrahedron that would not fail as long as the entire input set is not co-planar would be to substitute an arbitrary point that is not in the plane.

highest point above a facet f, i.e. the point whose projection onto the plane containing f has the furthest perpendicular direction. The point $hp(f_1)$ is found and this point, along with the three points in f_1, define the initial tetrahedron. The four facets of this tetrahedron are then added to the set of candidate facets t. Any points interior to the tetrahedron are deleted from P.

The iterative phase of "growing" the tetrahedron is then commenced. While P contains points, a facet f in t is examined to find $hp(f)$. If no such point exists then f is on the hull and can be moved from t to w. If a point is found, then the additional facets created by the edges of f and $hp(f)$ are added to t. Any points inside the tetrahedron constructed with f and $hp(f)$ are deleted from P, and f can now be removed from t as it is definitely not on the hull. Adding a new tetrahedron may make the polytope non-convex, as demonstrated in figure 3.5(a), where the new tetrahedron is shown dotted. This must be corrected to maintain hull convexity and is achieved by examining each of the three new facet/opposing facet pairs created, and when a non-convex join is found, the two facets are deleted from t and two new facets are created as shown in figure 3.5(b).

The process culminates once the set of points P becomes empty and there are no candidate faces left in t: w will then contain all facets making up the convex hull.

3.2.3 Planar algorithms

A lot of research has been completed on the convex hull of a planar point set. Generally this does not extend well to non-planar sets, so it is only summarised briefly here.

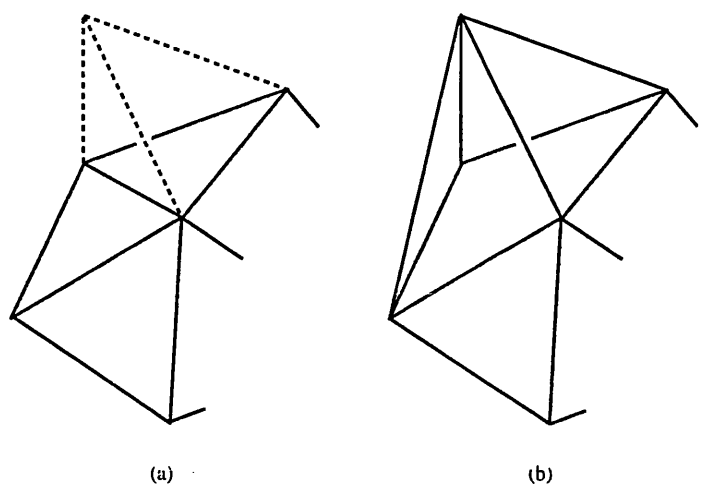


Figure 3.5: (a) Adding a point can make the hull non-convex. (b) The hull can be made convex by deleting two facets then creating two new facets.

Graham (1972) [30] gives an $O(n\,log\,n)$ worst-case time planar algorithm.

Jarvis (1973) [38] optimizes Chand and Kapur's d-dimensional gift-wrapping method [18] for two dimensions with time complexity of $O(nm)$ where m is the number of points on the hull.

Preparata and Hong (1977) [59] give $O(n\,log\,n)$ algorithms for two- and three-dimensional point sets that use a 'divide and conquer' approach. The convex hulls of the divided point sets are merged by finding the two lines that connect the two convex hulls and discarding all points inside the lines in the sets. This is then extended to three-dimensions. They suggest that questions of set separability and existence of linear decision rules are easily solved through determination of convex hulls.

Bentley and Shamos (1978) [5] give a 'divide and conquer' algorithm for an approximate convex hull of n points in two- or three-space in worst-case linear time. In three-dimensions the total running time is $O(n + k^2\,log\,k)$ and storage space is $O(k^2)$.

Akl and Toussaint (1979) [1] present an algorithm for two-dimensional point sets with worst-case complexity of $O(n \log n)$ and expected running time of $O(n)$ for n points. They discard points within a quadrilateral of extremal points on the hull. Of the remaining points, the angle between consecutive sets of three points (in the x dimension) is found, and the middle point is discarded if the angle is concave.

Preparata (1979) [58] gives a real-time on-line algorithm, which constructs the hull incrementally as it is given points, and for each point it has update time $O(\log n)$. Overall, it runs in time $O(n \log n)$.

Bentley, Faust and Preparata (1982) [3] give an algorithm for finding the approximate planar convex hull in $O(n + \frac{1}{E})$ time where E is a measure of the approximation. It splits the point set into strips, finds the set of extremes of each strip, and then finds the convex hull of the extremes set. It can be extended to d-space, however dimensions above three give large sets of extremes. This is improved upon by the work of Soisalonsoininen (1983) [70]. Stojmenovic and Soisalonsoininen then corrected errors in this work in 1986 [71].

Kirkpatrick and Seidel (1986) [43] give a solution for the two-dimensional convex hull problem using a variation on 'divide and conquer' that they have named 'marriage before conquest'. The problem is first divided into finding the upper and lower hull. For each of the upper and lower point sets, the set is recursively divided into two sets, and the 'bridge' part of the convex hull across the division is found. No merging is required, simply connecting the bridges end to end. Worst-case time is $O(n \log H)$ where H is the size of the output set.

Kao and Knott (1990) [41] give an efficient numerically-correct algorithm called *FastHull.* This works by discarding points that cannot be on the convex hull before using 'divide and conquer', then using a merge after solving the recursively divided problem. Extension to three-dimensions is briefly discussed but is largely left as an open problem. Worst-case time is $O(n \log n)$, and linear time performance is claimed for many kinds of input patterns.

Guibas, Salesin and Stolfi (1993) [32] give numerically robust algorithms for approximate hulls of inaccurate primitives.

3.3 New Research completed

There is a large amount of existing research on the convex hull of a set of points in any number of dimensions, without, as far as I am aware, the problem of computing convex hulls of set-theoretically defined models being addressed. A method of finding the convex hull of a semi-algebraic set that is built around an adaptation of an existing point set convex hull algorithm for set-theoretic models is proposed. This has been implemented using the sVLIs solid modelling kernel and results are given. The future work section discusses an original multi-dimensional set-theoretic modelling method of finding the convex hull that was originally suggested to me by Woodwark [78].

3.3.1 Adaptation of an existing point-set algorithm

Most existing convex hull algorithms compute the convex hulls of point sets. Using one of these algorithms for a set-theoretically defined model would entail finding a point set that represents the model, and using this point set as input for the convex hull algorithm. Any model that contains curved objects can only be approximated by a point set, although an approximate convex hull is still useful for many applications such as collision detection and path planning. If the model only contains polytopes then it is possible to generate the exact convex hull. Convex hull algorithms usually find the convex hull as a point set. For use in set-theoretic modelling it would be desirable to convert this into an intersection of planar half-spaces.

A method for computing the convex hull of set-theoretically defined models has been developed, which has three distinct phases:

1. generate a set of points that enclose the model,
2. using a point-set convex hull algorithm, find the convex hull of those points,
3. represent the resulting convex hull set-theoretically using planar half-spaces.

Generating points that enclose the model

A set of points that enclose the set-theoretically defined model must first be generated, so that the convex hull of these points can be found. Three methods of generating these points have been used: for polyhedral input models the exact points required are found from the vertices of the model, while for non-polyhedral models two methods generate points that approximate curved models by the following means:

1. recursively divide the model and use the corners of the smallest boxes.
2. facet the model and use the corners of the facets.

The vertices of polyhedral models are found by using a three-dimensional vertex finder written by Bowyer, Eisenthal and Wise as part of the sVLIs ^{m} project [12]. This works by recursively dividing the model until the sub-model under consideration contains three or less planes. The point of intersection of the three planes is then determined analytically, and this point is added to the list of vertices. The process continues for all sub-models in the divided model tree. The convex hull created from these points is exact within the accuracy of floating point arithmetic.

For the first point-generation method for non-polyhedral models, the model is first recursively divided with no overlap of the divided model's boxes, i.e. the box-swell factor is set to zero. The divided model's tree is traversed, and for each surface leaf sub-model of the tree each of the corner points of the sub-model's box are examined: if a point is not within the solid part of the model it is saved for input into the convex hull algorithm. Corner points of boxes which are in solid parts of the model do not need to be considered by the convex hull algorithm because they are surrounded by other sub-model boxes' corner points that are not in solid, assuming that the model is closed within its box. To demonstrate that this is true, consider a point that is in solid and let its adjacent points be the points that are directly connected to it by the edges of the divided model. Consider the case where all of the adjacent points are in air. All of the adjacent points will be corner points of either the current box or an adjacent box. In either case, the box of which they are a corner must be a surface box, as it has one corner in solid, and one corner in air. The box's corner points, and in particular the adjacent points, will be included in the points that are output, and since they are adjacent to the solid point the latter will be discarded by the convex hull algorithm. If any of the adjacent points are solid, then the adjacent boxes that have them as corners may also be solid. These points and boxes do not need to be output for the convex hull algorithm, because they will eventually be surrounded by other points in air which will be output by the same mechanism.

Some of the points generated will be duplicates as they are corners of more than one sub-model's box. These duplicate points are removed before the point set is output. (Note that a simple scheme of considering only the corner of a box that has the minimum value in all axes would not work as the boxes have different sizes)

The convex hull created by this method is an approximation, nevertheless it is guaranteed to contain the exact convex hull of the model - a property which is useful for applications such as collision detection and avoidance.

The second method of generating points for non-polyhedral models starts by faceting the model, i.e. creating polygons that represent approximately the model's surface (these are most frequently used for display purposes). The corner points of each facet are generated for input to the convex hull algorithm. In a similar way that box corners in the first method may be duplicated, the corner points of facets may be shared by several facets: for efficiency, such duplicate points are removed. The convex hull resulting from these points is a close approximation to the exact convex hull.

The convex hull algorithm

Allison and Noga's convex hull program 'tetra' [2] (described earlier in section 3.2.2) has been used to find the convex hull of the point set. The tetra program takes as its input a point set, and outputs both the points on the convex hull and facets as indices into the list of points. I have adapted the program so that in addition to producing the points on the convex hull and the facets of the convex hull it also generates a point pt_{in} that is within the convex hull; this point is the centroid of the initial tetrahedron that is determined as the first step in the convex hull algorithm, after the point elimination pre-processing step (see section 3.2.2, page 24). This point is used when converting the point set into set-theoretic representation to determine which side of each facet is solid.

Converting the point set into an intersection of planar half-spaces

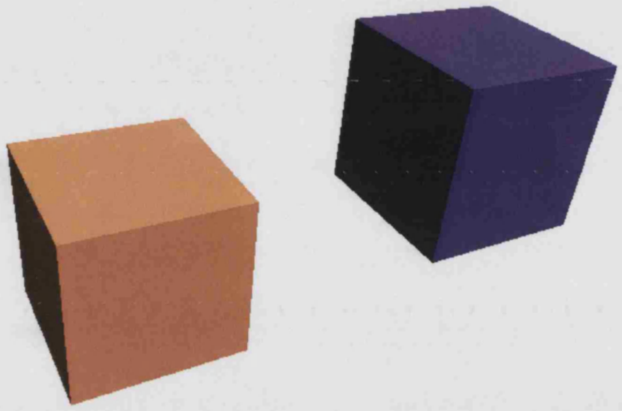
The resulting convex hull generated by tetra as a point set and facets is converted into a set-theoretic representation. A planar half-space is created for each facet that is coincident with the facet and which has its solid side towards the point within the hull (pt_{in}). The set-theoretic model of the convex hull is then created as the intersection of all the planar half-spaces.

3.3.2 Results

The convex hulls of several polyhedral and non-polyhedral models have been computed by using the three point generation methods proposed (one for polyhedral models and two for non-polyhedral models). Results for polyhedral models are shown in figures 3.6 to 3.8, where the points have been generated by finding the vertices of the model. The convex hulls found for these models are exact.

Allison and Noga's tetra convex hull program fails to create the convex hull for some point sets that have been generated, resulting in the error message "attempt to delete a point known to be on the hull". To date it has not been possible to ascertain the reason for these failures. There is no correlation between the complexity of the point sets and the failures; for example the points generated from the model shown in figure 3.9 are relatively simple and yet cause the error, whereas more complicated point sets also cause the error. For some point sets tetra can successfully find the convex hull, but if some of the points in the point set are translated by a small amount then tetra fails to find the convex hull of the new point set.

The convex hulls of three non-polyhedral models have been calculated by generating points by using the two methods of using the corner points of sub-model's boxes in a divided model and of using the corner points of the faceted model's facets. Example results for these models are shown in figures 3.10 to 3.18. The division resolution and facet factors for these examples have been chosen so that the number of points that are sent to the convex hull program tetra is approximately the same for each model for both methods (exact figures can be seen in table 3.1). The faceting method creates convex hulls which are considerably closer to the original model's surface, although the hull is approximate and may be partly inside the original model.



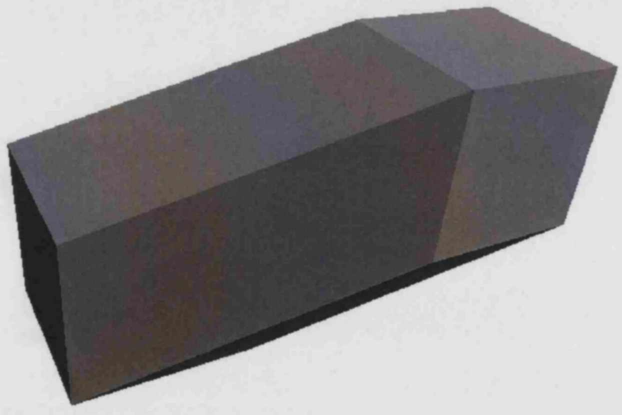


Figure 3.6: 'Two cuboids' polyhedral model and the convex hull produced.

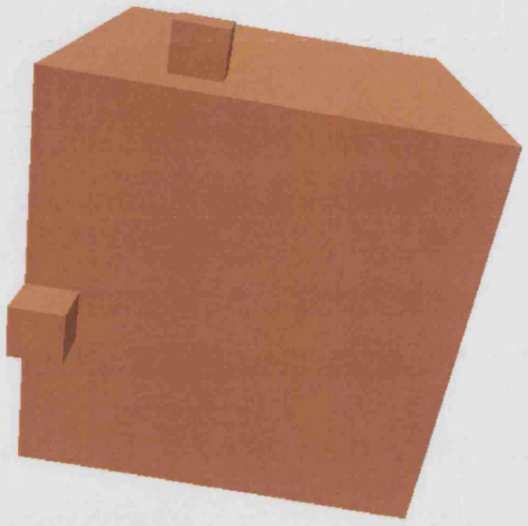
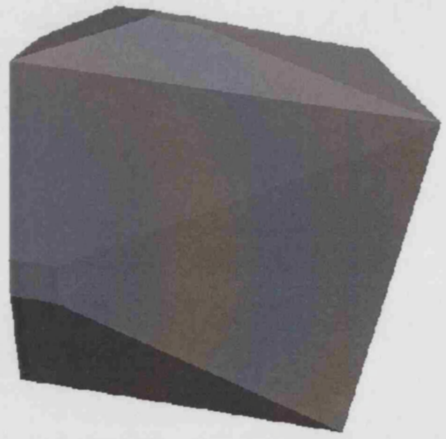



Figure 3.7: 'Cuboid unioned with smaller cuboids' polyhedral model and the convex hull produced.

Figure 3.8: ‘Simple robot arm’ polyhedral model and the convex hull produced.

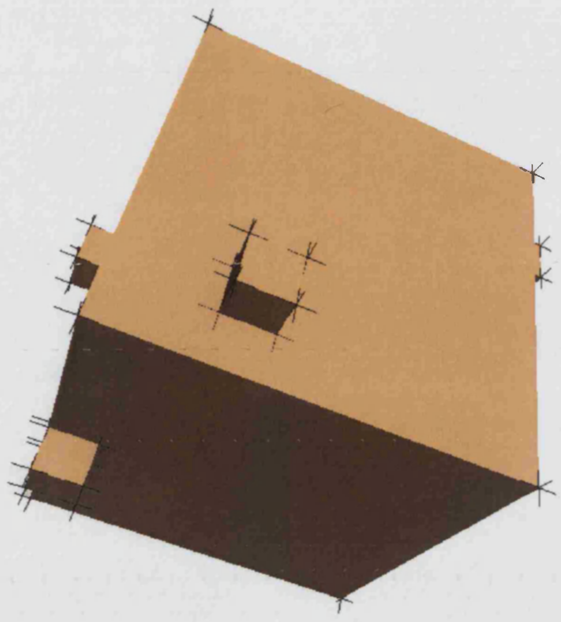


Figure 3.9: Model for which the convex hull program tetra fails to create the convex hull.

Model	Division resolution					Facet factor				
	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	1	10^{-1}	10^{-2}	10^{-3}	10^{-4}
Robot	78	154	471	*1342*	4362	*519*	2501	9703	25883	25883
Wristwatch	187	740	2865	*8249*	20285	1339	2586	3415	3897	*4108*
CSG	43	126	783	*2425*	8003	*2075*	4675	5701	6798	7514
Sphere	146	458	2306	8730	11242	896	896	896	896	896

Table 3.1: Number of points generated for the convex hulls for the model division and model faceting point-generation methods for non-polyhedral models.

Table 3.1 gives results for the number of points generated for the two point-generation methods of dividing the model (for various division resolutions) and of faceting the model (for various facet factors, which are measures of the accuracy of the resulting faceted model and result in the model being divided more finely before facets are created). The number of points is shown after duplicate points have been removed. The values in italics denote the results shown in figures 3.10 to 3.18. The division resolutions and facet factors are not directly comparable because the division resolution is the ratio of the volume of the smallest possible

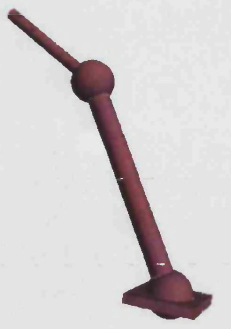


Figure 3.10: 'Robot arm' example model.

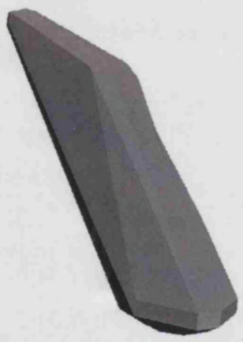


Figure 3.11: Convex hull of 'Robot arm' model from points generated by model division with division resolution of 10^{-5}.

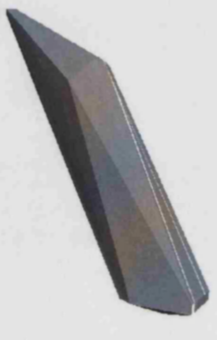


Figure 3.12: Convex hull of 'Robot arm' model from points generated by faceting the model with facet factor of 1.

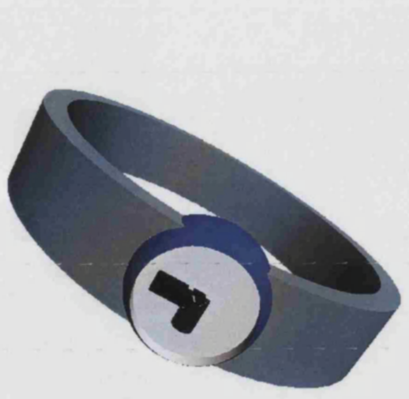


Figure 3.13: ‘Wristwatch’ example model.

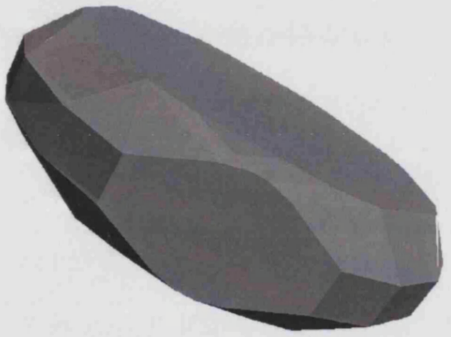


Figure 3.14: Convex hull of ‘Wristwatch’ model from points generated by model division with division resolution of 10^{-5}.

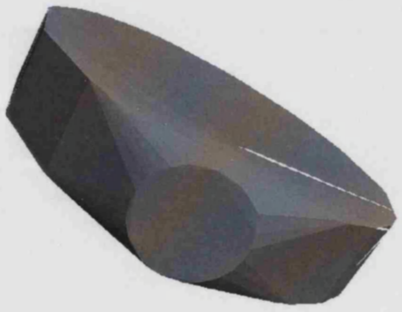


Figure 3.15: Convex hull of ‘Wristwatch’ model from points generated by faceting the model with facet factor of 10^{-4}.

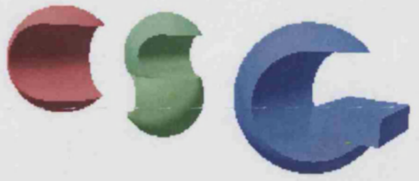


Figure 3.16: 'CSG' example model

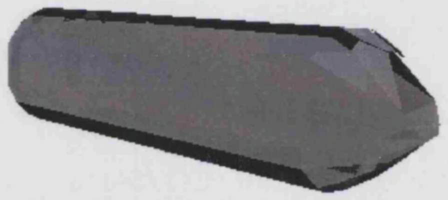


Figure 3.17: Convex hull of 'CSG' model from points generated by model division with division resolution of 10^{-5}.

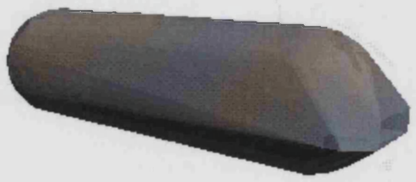


Figure 3.18: Convex hull of 'CSG' model from points generated by faceting the model with facet factor of 1.

Model	Division resolution					Facet factor				
	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	1	10^{-1}	10^{-2}	10^{-3}	10^{-4}
Robot	0.00	0.06	0.17	*0.42*	1.52	*0.32*	1.26	4.80	10.26	10.47
Wristwatch	0.05	0.15	0.77	*1.81*	4.59	0.55	1.54	1.42	1.61	*1.70*
CSG	0.05	0.06	0.27	*0.89*	2.93	*0.94*	1.89	2.17	2.59	2.88
Sphere	0.05	0.16	0.99	3.92	4.91	0.42	0.42	0.42	0.42	0.42

Table 3.2: Time in seconds taken to generate points for the convex hulls for the model division and model faceting point-generation methods for non-polyhedral models.

sub-model box to the volume of the original model's box, and the facet factor is a measure of facet accuracy. The number of points found is the same for the 'Robot' model at facet factors 10^{-3} and 10^{-4} because the faceting algorithm produces the same facets at these facet factors.

Table 3.2 shows the time taken in seconds to generate points for the two point-generation methods for each model at several division resolutions and facet factors. The values in italics denote the results shown in figures 3.10 to 3.18. The times are measured as CPU time taken as an average of three executions of the program on a Windows 95 based computer with a 266MHz K6-2 (Pentium II class) processor and 64Mb of RAM . It takes a similar amount of time to perform division as it does to facet the model to generate the same number of points for a given model. As the division resolution or facet factor is made finer, the number of points generated increases but the time taken to generate the points does not increase linearly with the number of points. This is usual for adaptive binary spatial division as the division time is affected by other factors.

3.4 Future work

3.4.1 Improvements to the adapted point-set algorithm

The 'tetra' convex hull program that has been used to find the convex hull of a set of points generated from a set-theoretically defined model has problems handling some points sets as identified on page 32. To ensure that this convex hull method can be used on all models it would be important to either identify the cause of this problem, or to use an alternative point-set convex hull algorithm.

The data structure that is currently used to store the points found from the set-theoretically defined model before they are sent to the point-set convex hull algorithm is an unsorted linked list of points. It is necessary to remove duplicates from this list before the points are sent to the convex hull algorithm. It would be more computationally efficient to use an alternative data structure that supports fast point insertion without inserting duplicate points, such as an ordered binary tree.

3.4.2 Multi-dimensional technique for convex hull computation

The framework of this technique was suggested to me by Woodwark [78], who proposed a multi-dimensional method of finding the convex hull of a set-theoretically defined model. Given such a model and an infinite number of planar half-spaces which may be in any position in space and have any orientation, the convex hull of the model is the intersection of all planar half-spaces that satisfy the condition

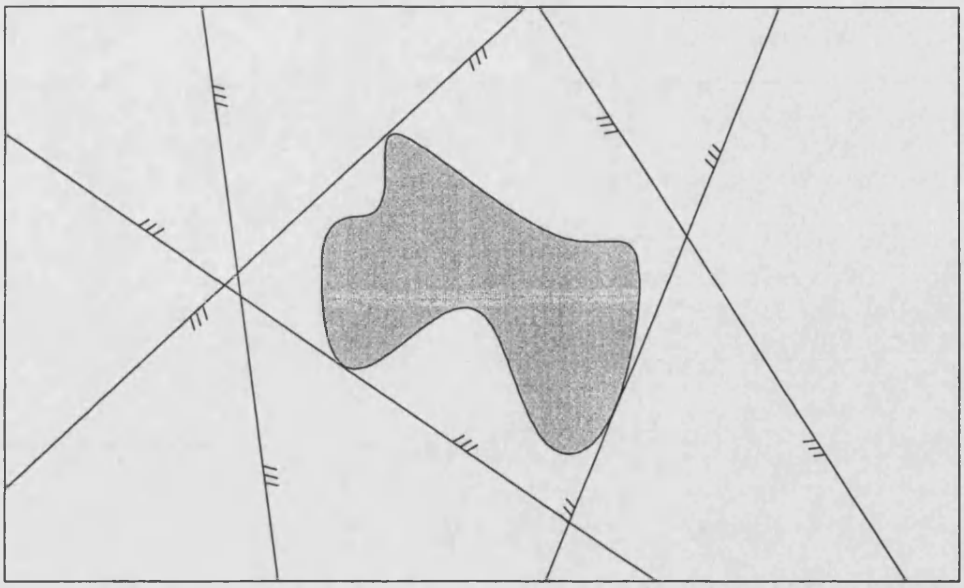


Figure 3.19: Example for multi-dimensional convex hull method.

$$half\text{-}space \cap model = model \quad (1)$$

i.e. all planar half-spaces which completely enclose the model. Figure 3.19 illustrates this with a two-dimensional example model, and shows some of the planar half-spaces that satisfy this condition. As this is a two-dimensional model the planar half spaces are lines. Generally with this method there is some redundancy in the planar half-spaces, since half-spaces that do not touch the surface of the model are unnecessary. The half-spaces would have three degrees of freedom if the model is a three-dimensional model, or more generally d degrees of freedom for a d-dimensional model.

This lends itself to a multi-dimensional implementation, which could use the sVLIs ^{m} multi-dimensional CSG modeller [12]. A one-dimensional diagram, figure 3.20, is used to illustrate how this would work. The one-dimensional model consists of two intervals I_a and I_b, combined with the union operator. A multi-dimensional space is created containing the original model's dimensions, the dimensions of translation of the model in each of the original dimensions, and the rotation of the model. In the example this space contains the model dimension, x, the translation dimension, x_t, and a rotation dimension which has only two values as a one-dimensional planar half-space can only be rotated into two positions, so that the solid side is either at lower values of x or at higher values of x. Within this space, a multi-dimensional model is created. This model is solid at the points where condition (1) is satisfied for a half-space at the position specified by the position in the multi-dimensional space. In the example half-spaces with x values lower than the lowest interval with the solid side of the half-space in the direction of increasing x satisfy condition (1) (shown as the region P_1), as do half-spaces with x values higher than the highest interval with the solid side of the half-space in the direction of decreasing x (shown as the region P_2). This multi-dimensional model is next projected down into the original model's space, shown as the regions R_1 and R_2. To enumerate this projected model, adaptive recursive spatial division could be used to find solid, air and surface areas, with finer division used to make the results more accurate. The convex hull would then be represented by the regions of the projected model that are air, i.e. the convex hull would be the complement of the projected model.

3.5 Conclusions

A method for computing the convex hull of a set-theoretically defined model has been presented. This method consists of generating a point set from a model and then using an existing point-set algorithm to find the convex hull of the point set. The point set that is created by this to represent the convex hull can then be converted into a set-theoretically defined model.

Different methods of generating the point set have been implemented: one for

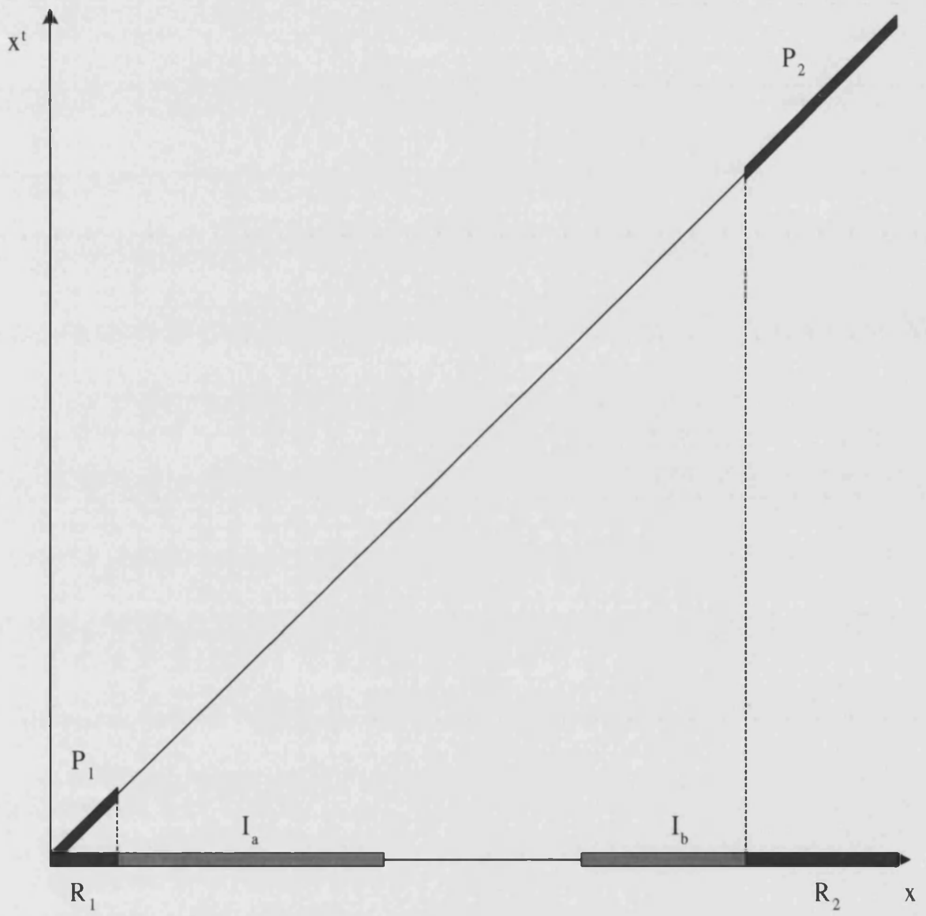


Figure 3.20: Projection from multi-dimensional space to original model space for convex hulls.

polyhedral models and two for non-polyhedral models. The method for polyhedral models finds the exact convex hull. The two methods for non-polyhedral models use recursive division and faceting, and create approximate convex hulls. Of these two methods the faceting method was found to produce convex hulls closer to the original model, although these are not guaranteed to completely enclose the model. Convex hulls generated by the recursive division method for non-polyhedral models will always enclose the original model.

Chapter 4

Connected component labelling

4.1 Introduction

Connected component labelling is used in model analysis to determine whether two points of a model are connected, and also to count the number of separate components of a model. One application of connectivity information is component analysis. An example of this could be that an engineer has created a computer model of engine components and wants to be sure that fluid will not leak from one engine chamber to another. Another possible situation could be that after creating the model the engineer wants to check how many separate components there are to identify design errors such as inadvertently splitting a component into two separate objects.

Boundary representation modellers have connectivity information explicitly stored in the data structure and so, in theory, it should be easy to find the connectivity information for a boundary representation model. In set-theoretic modelling however, there is no straightforward way of extracting the connectivity information because the model is unevaluated, i.e. the structure of the set-theoretic tree of boolean operators and primitives has no direct relation to the connectivity of the model.

There are several connectivity problems with differing levels of complexity. A relatively simple connectivity problem is to determine if two points in three-dimensional space are connected by a path passing only through the 'solid' parts of a model or only through the 'air' parts of the model (this would solve the foregoing example above of checking that fluid cannot pass between two chambers: if the two chambers are connected by a path passing through 'air' then the fluid will leak). A more difficult problem is to label the connected components and hence find the number of separate components within a model. By labelling the components the point-connectivity problem can be answered by examining the labels of the parts of the model that the two points lie in, although the path between the two points will not be known.

4.2 Literature survey

Research in the field of connected component labelling uses various approaches and data structures such as 2^n trees and binary trees, geometrical objects, voxel images and line segments. The first two data structures are reviewed in detail in this section, and the latter two are only briefly mentioned below as they are not relevant to the current research because the methods used are not compatible with the data structure of the sVLIs geometric modeller. Much connected component research has been done in the area of vision analysis [36].

Voxel image algorithms such as Lumia's [51] and Thurjfell, Bengtsson and

Nordin's [73] use two phases: firstly scan each slice, row and column of an image and give adjacent voxels the same label, and secondly re-label the model using label equivalences that are found in the labelling phase. This process is covered in more detail in the explanation of Samet's work [64], given below. Work on connected components for line segments, such as that by Lopez and Thurimella [49], is concerned with the special data structure of sets of line segments, which can be applied to the fabrication of VLSI electronic circuits.

4.2.1 2^n trees and binary trees

Much of the work on 2^n trees and binary trees is within the field of computer graphics, where images are often stored using these tree structures. The internal nodes of these trees are considered to be grey and the leaf nodes are either black or white - these leaf nodes can be considered to be equivalent to the solid and air nodes (see section 2.2) of a spatially-divided model in set-theoretic solid geometrical modelling. There is no equivalent to surface leaf nodes in set-theoretic modelling for trees representing images.

One connected-component labelling approach is *neighbour-visiting* [64, 23] in which all of the nodes in the image are visited in a certain order, and when a black node N is encountered each of its' neighbours are visited to see if they are black. If any neighbour is black, then node N and that neighbour are given the same label (if they are both already labelled, the two labels are added to a list of equivalent labels). Another approach visits each node in turn and keeps track of an *active border* [66, 67] which stores the colour of nodes adjacent to the border; this makes it unnecessary to visit the neighbours of each node. However as the number of dimensions increases the size of the active border data structure can become excessive.

Neighbour-visiting algorithms use two approaches to find neighbours: *top-down* or *bottom-up*. The top-down approach [45] uses the coordinates and size of the current node to compute the location of a point in the neighbouring node. The tree can then be walked from the root downwards to find the node which contains that point. The bottom-up approach (used for example by Samet [64]) makes use of parent pointers in each node to walk up the tree in order to find the node that contains both the current node and its neighbour. The tree can then be walked down, mirroring the nodes chosen when walking up the tree, to find the neighbouring node of equal or greater size.

Samet (1981) [64] presents an algorithm for labelling the connected components of an image represented by a quadtree. The quadtree is traversed in postorder i.e. the children of a node are visited before the node itself, and children are always visited in the fixed order Northwest, Northeast, Southwest, Southeast. As each leaf node is visited, its Eastern and Southern neighbours of equal or greater size are found, and all children of those neighbours adjacent to the leaf node are examined to see if they are solid. When two adjacent solid nodes are found, they are given the same label; if two adjacent solid nodes are already labelled, their labels are added to a 'label equivalence' list, and the label equivalences are merged after the tree has been traversed.

Samet and Tamminen (1985) [66] give algorithms for linear quadtrees that do not require the calculation of the neighbour of a node. These algorithms are used to calculate perimeters, label connected components and to calculate the *Euler number* for two-dimensional images. The Euler number, or genus, is the difference between the number of connected components and the number of *holes* (a hole is an air component completely enclosed by a solid component). These algorithms use an active border to keep track of information relating to nodes that have already been visited, which reduces the computational complexity of

Samet's original approach [64] but requires more storage (for the active border).

Samet and Tamminen (1988) [67] introduce a multi-dimensional connected component algorithm that uses active borders in a similar way to their earlier work in 1985 [66]. The running time of the algorithm is 'almost' linear with respect to the number of nodes in the 2^n tree, and worst-case storage requirements are $O(b)$ where b is the number of solid nodes.

Samet (1989) [65] presents algorithms for finding neighbours in octrees using a bottom-up approach; this technique is useful for neighbour-visiting in connected-component labelling algorithms.

Dillencourt, Samet and Tamminen (1992) [23] give an algorithm for connected-component labelling for a variety of image representation schemes, including quadtrees and binary trees. This algorithm labels adjacent solid nodes in the image with the same label by scanning the image in a similar way to [64]. Two methods are used to reduce storage requirement: an intermediate file is used between two passes and labels are re-used.

A major part of the computational effort for connected component algorithms is keeping track of active elements (active elements are those elements which have been scanned but have at least one neighbour that has not been scanned). The order in which the image is scanned is also important. Dillencourt, Samet and Tamminen define three categories of scanning orders: *admissible*, *weakly admissible* and *inadmissible.*

Admissible scanning orders limit the number of active image elements. For a scanning order to be admissible two criteria must be met: firstly each element must be processed only once, and secondly all neighbours of each

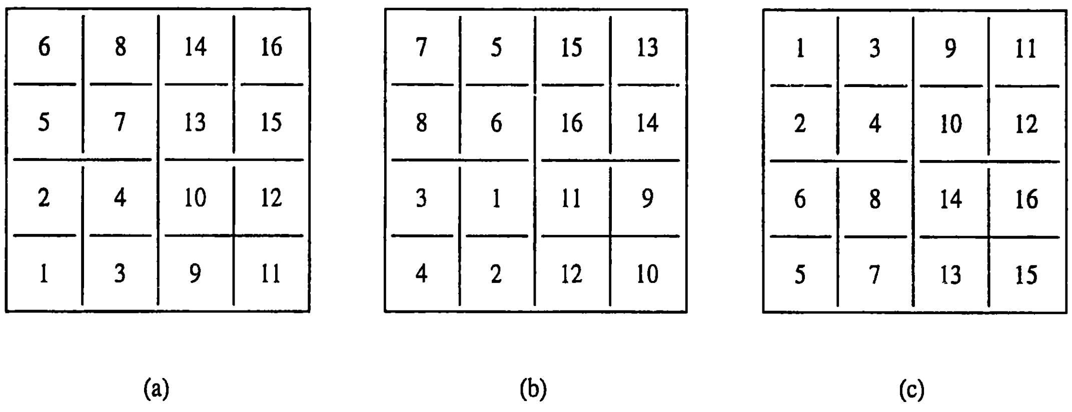


Figure 4.1: Example scanning orders for two-dimensional binary trees: (a) admissible, (b) inadmissible, (c) weakly admissible.

image element in a set of *distinguished directions* will either have already been scanned or will not exist (i.e. the image element is on the edge of the image). The distinguished directions consist of one direction from each *direction pair* (i.e. the North-South pair or the East-West pair).

Weakly admissible scanning orders also ensure that each image element is processed once and that when processing any image element, all of its neighbours in at least one direction of every *direction pair* (such as North-South or East-West) either do not exist or have been scanned already. The directions can be chosen differently for each image element. The only difference between admissible orders and weakly admissible orders is that the neighbours must be in a set of distinguished directions. All admissible orders are also weakly admissible.

Inadmissible scanning orders are all scanning orders that are not weakly admissible nor admissible.

An example of an admissible order for binary trees would be to first scan the child in the minimum direction of each axis and is shown in figure 4.1a. The distinguished directions for this example are South and West. Inadmissible and weakly admissible orders for binary trees are counter-intuitive; for example, alternately choosing the child with the lowest value in all axes, then choosing the child with the highest value in all axes, would produce the inadmissible order of figure 4.1b. A weakly admissible scanning order is shown in figure 4.1c, which could be produced by applying the rule: choose the least child in each direction, unless the current node is a maximum node in the North-South direction, in which case choose the child node with the maximum value in the North-South direction to be the next node to be scanned.

4.2.2 Connected components of geometrical objects

Edelsbrunner et al (1984) [25] propose a connected component labelling method for horizontal and vertical line segments and for rectangles in a two-dimensional plane. It can be performed in $O(n\,log\,n)$ time and $O(n)$ space. In this approach a vertical line is swept across the objects and a connected component data structure is stored which is updated whenever the sweeping line either comes onto, or comes off, a horizontal segment, or when it intersects a vertical segment. For rectangles there are two cases which need to be handled: when the sweeping line meets the left end of a rectangle or when it meets the right end of a rectangle. They also discuss a method for rectilinear objects with dimensionality higher then two, which does not use a sweeping approach, but instead uses a method of intersecting d-dimensional rectilinear objects with one another. This can be processed in $O(n\,log^d\,n)$ time and requires $O(n\,log^{d-1}\,n)$ space where d is the dimensionality of the objects.

Canny (1988) [17] has proposed a 'roadmap' algorithm which finds the one-dimensional skeleton of a semi-algebraic set, called the roadmap of the set. The algorithm works in single exponential time. The relevance of this to connected component labelling is that if a set is connected then its roadmap is connected, hence the roadmap of a set can be used to determine whether or not two points lie in the same connected component of that set. In [16] (1993) Canny removes the restrictions that the set must be compact and 'in general position'.

Heintz, Roy and Solernó (1994) [33] have also used roadmaps in an algorithm for finding the connected components of a semi-algebraic set. Their method does not require that the set be in general position, and does not need a 'Whitney stratified input' [17].

4.3 New Research completed

I have suggested two methods to determine whether or not two points are connected in a set-theoretically defined model which uses intersection of line segments; these methods are described in section 4.3.1, below.

Furthermore, the connected component labelling problem has been approached by analysing the binary tree created by recursive spatial sub-division of a sVLIs model in order to find connected *boxes,* as described in section 4.3.3. Connected boxes are given the same label, and the number of different labels gives the number of components. I have adapted Samet's connected component labelling algorithm for two-dimensional quadtrees representing two-dimensional monochrome images, so that it works with binary trees representing three-dimensional set-theoretic models. Finally, this connected component labelling method is adapted to use an analytical method of finding the connectivity of polyhedral models, as described in section 4.3.5.

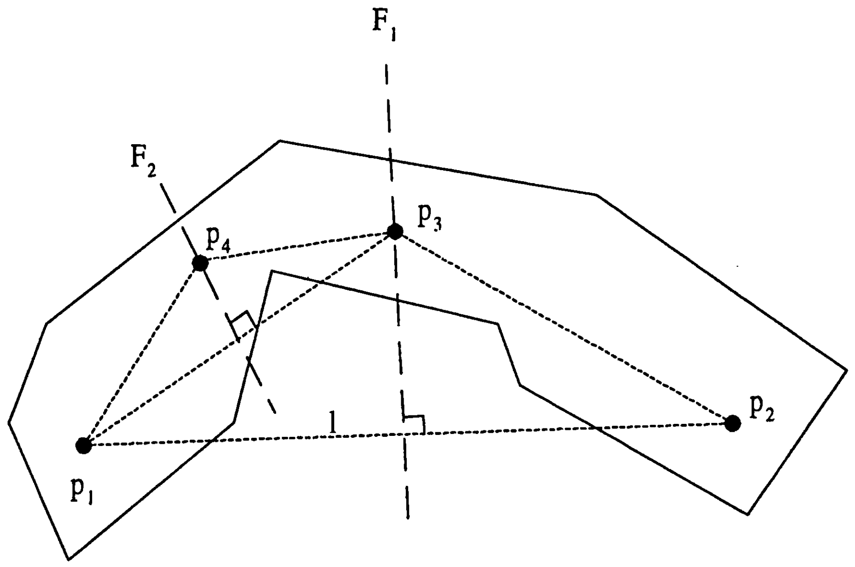


Figure 4.2: Example object for divide and conquer solution to point connectivity query.

4.3.1 Point connectivity

This section briefly describes a heuristic for point connectivity that is possible to implement for any modeller offering point membership and line-intersection (ray-trace) queries. It may not always find a solution, but any solution found is guaranteed to be valid. A 'divide and conquer' approach is used that divides the line between the points until a series of line segments have been found that do not pass through any part of the model. All examples in this section are two-dimensional for clarity within diagrams, however the method suggested has been used in more than three dimensions using the multi-dimensional modeller sVLIs ^{m} [12].

It is assumed that the two points between which connectivity information is required are in solid objects and are denoted by p_1 and p_2, and also that l is the line segment (p_1, p_2) as shown in figure 4.2. If l does not intersect with any surfaces, then the two points must be connected. If l does intersect with some

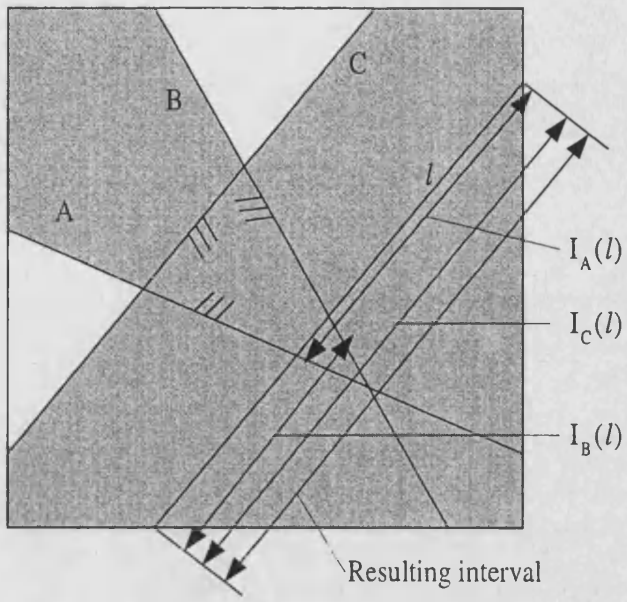


Figure 4.3: The intersection of the line segment l with a sub-model $(A \cap B) \cup C$, i.e. $I_R(l)$ is $(I_A(l) \bigcap I_B(l)) \bigcup I_C(l)$, which covers the entire line segment.

surface, then the plane F_1 that is the perpendicular bisector of l is found. A point p_3 is found that lies in the plane F_1 and is also in solid. The process is now applied recursively to the pairs of points (p_1, p_3) and (p_3, p_2) until either the points are connected by a path through solid, or a bisecting plane is found that is completely in air, so the two points are known to be disconnected. If neither of these two conditions are met within a given number of iterations the process must be terminated, as the connectivity relationship between the points cannot be established. The planes and points found for a two-dimensional object are shown in figure 4.2. In this example only one more recursion is needed to find p_4 between p_1 and p_3.

The proposed method needs to be able to determine whether or not l intersects with any surface in a model, and also needs to be able to find a point on a plane that is in solid. The first of these requirements - the line-segment intersection mechanism - is available within the sVLIs geometric modeller in the form of ray-tracing operations, which are more efficient if the model has been divided. The

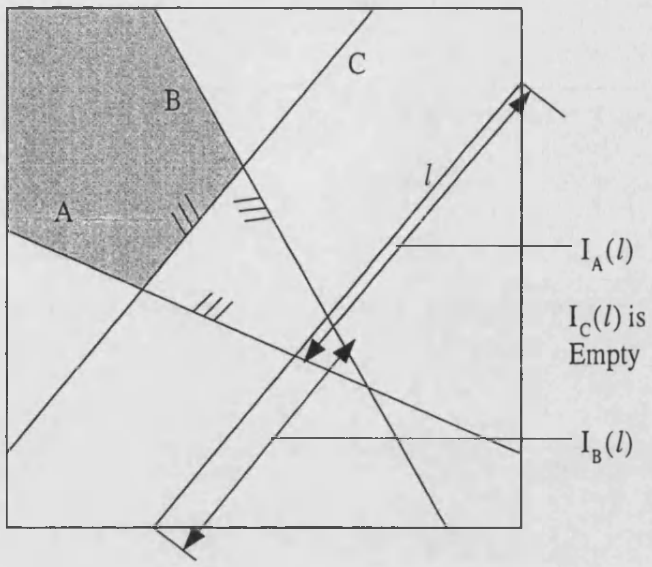


Figure 4.4: The intersection of the line segment l with a sub-model $(A \cap B) \cap C$, i.e. $I_R(l)$ is $(I_A(l) \bigcap I_B(l)) \bigcap I_C(l)$, which is the empty interval.

first leaf sub-model that l intersects is found by comparing the model's box and sub-model's boxes with l. For each primitive X in a leaf sub-model the interval $I_X(l)$ of parametric values of l for which l intersects X is computed. These intervals are then combined in the same way as the primitives are combined in the set-theoretic expression that defines the set within the leaf sub-model's box, resulting in an interval or series of intervals $I_R(l)$ of the intersection of the set tree with l. Examples of the computation of this interval are given in figures 4.3 and 4.4. If $I_R(l)$ covers the entirety of l (for example figure 4.3) or is the empty interval (for example figure 4.4) the model tree traversal is continued to find the next leaf sub-model that l intersects, and the interval examination is performed again. Alternatively if the interval does not cover the whole of the sub-model's box, or a series of intervals has been found, then l must intersect with the set and the first primitive that l intersects is found at the boundary of the first interval. If l does not intersect with any of the leaf sub-models then it is known not to intersect the sub-model.

The second requirement of a modeller that is to be used for this point connectivity technique is to find a point that is on the bisecting plane F and that is also in the solid region of the set. A recursive division technique or a Monte Carlo technique can be used to do this. For the recursive division technique, the concept of two sub-models being *directly connected* is required. Two sub-models are directly connected if a line-intersection query on the line segment between the centroids of the two sub-models' boxes results in no intersection with the surface of the sub-model. To find a point in F and in solid the sub-models that F passes through are examined, with three possibilities:

One or more of the sub-models are completely solid. The point on the plane that is closest to the centroid of that sub-model's box is used. If more than one solid sub-model is found, and these sub-models are not *directly connected*, then the recursion must be performed in turn on the points found in each of those sub-models' boxes.

The plane F passes through only surface sub-models. The sub-models are divided until a solid one is found.

All of the sub-models that F passes through are air. The two points are known not to be connected.

An alternative approach to point connectivity

An alternative method for establishing point connectivity has also been originated where, instead of finding one perpendicular bisector, two planes $F_{1,1}$ and $F_{2,1}$ are found. $F_{1,1}$ and $F_{2,1}$ are perpendicular to the line segment and pass through the points where the ray emerges from the surface. The point $p_{1,1}$ is found, where this points lies in solid and on plane $F_{1,1}$. Point $p_{2,1}$ is found in a similar manner. A ray is now fired from $p_{1,1}$ to $p_{2,1}$ and the process is repeated until

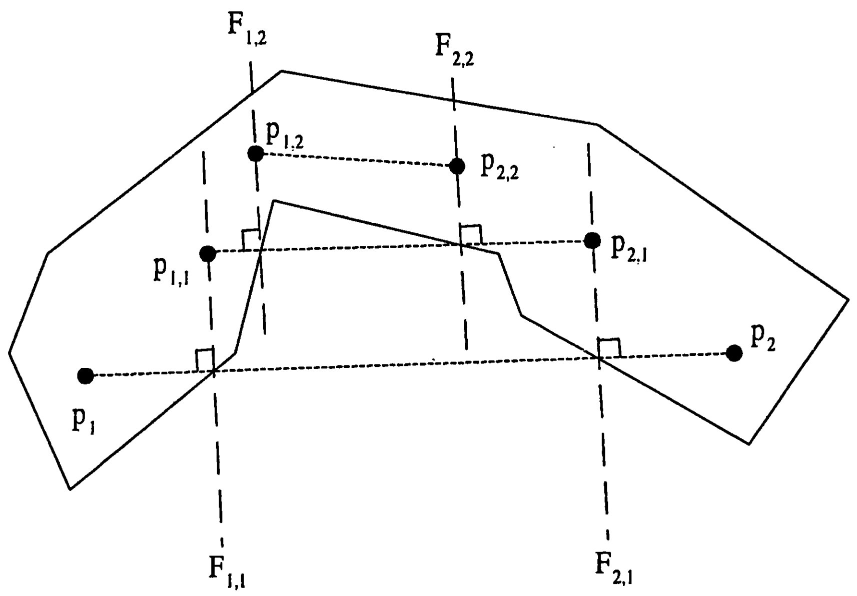


Figure 4.5: Example object for alternative divide and conquer solution to point connectivity query.

either a connection has been established, or a bisecting plane is found that is completely in air, in which case the two points are known to be disconnected. As with the previous method the process must also be terminated if neither of these two conditions are met within a given number of iterations, in which case the connectivity information for the points remains unknown. Figure 4.5 shows the same two-dimensional object as that of figure 4.2, with the points and planes found by this alternative method. The alternative method required two iterations to determine the connectivity information for this example.

Suitability of the techniques

Both of the preceding techniques can fail to provide connectivity information if a bisecting plane has been divided to the division limit and only surface boxes result. In this situation the initial division resolution could be made finer, but that may not always lead to a solution. For instance it is possible to construct two disconnected objects between which there is no dividing plane that is completely in air; for such objects these techniques will terminate with unknown connectivity. However the techniques would be useful as heuristics if they were part of a system where the exact answer was not required.

The techniques are inefficient if two objects are very close but are not connected. In this situation a large number of iterations would be required if the gap between the objects does not fall in one of the splitting planes found in an early iteration.

The techniques are therefore suitable for confirming that two points in solid parts of the model are connected, for example in model analysis where there is a requirement that two points in the model are connected. Because of the nature of set-theoretic modelling, to check that two points in air parts of the model are connected, these techniques can be applied to the complement of the model.

4.3.2 Results

Bowyer [11] has created a multi-dimensional implementation of my point connectivity technique using the sVLIs ^{m} [12] multi-dimensional set-theoretic modeller. The configuration space of a two-dimensional model that is able to rotate (i.e. it has one degree of freedom) is three-dimensional and is shown in figure 4.6. This modeller is used in a path planning situation where a configuration space [50] has been generated for a three-dimensional *nomad* which can translate and rotate but must not clash with a three-dimensional *obstacle*; this configuration space cannot be illustrated as it is six-dimensional. The start position and final position of the nomad are two points in the configuration space, and the point connectivity technique presented here is used to find out whether or not there is a connection between these two points: if there is then the series of line segments between the

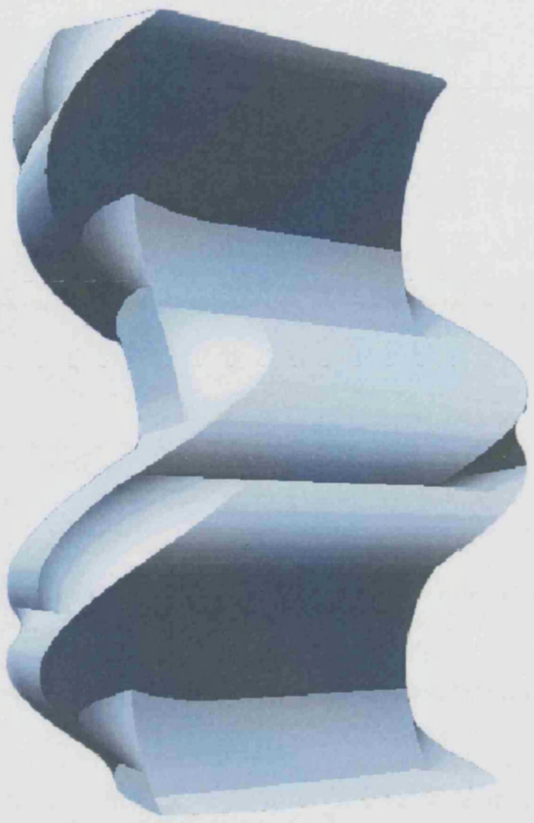


Figure 4.6: The configuration space of a two-dimensional model with one degree of freedom.

intermediate points in the configuration space gives a path in the model space.

The start and final positions of a nomad (the brown union of two tetrahedra) and an obstacle (the blue cube) are shown in figure 4.7, and figure 4.8 shows a point found on the path between these two positions. The small green cube is a graphical flag used by the configuration space mapping program to show that the positions of the nomad and the obstacle are valid, i.e. they do not overlap. An animation of the nomad following the path that has been generated is available on Bowyer's website [9]. The time taken to find a path for this problem was approximately two minutes on a 400MHz Pentium II running Linux.

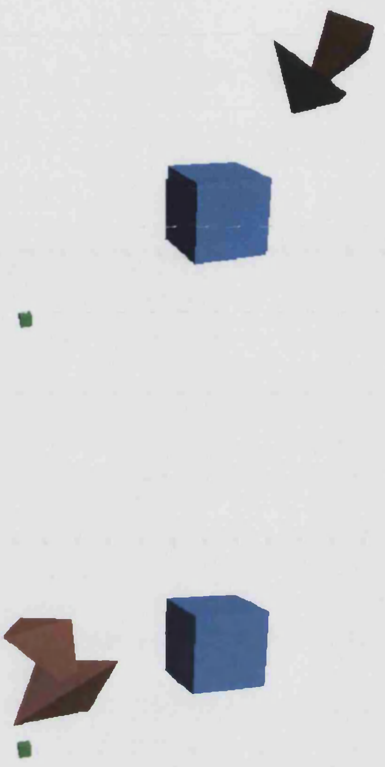


Figure 4.7: The start and final positions of a nomad (the brown union of two tetrahedra) either side of a simple obstacle (the blue cube). The green cube is a collision flag and is not part of the geometry of the problem.

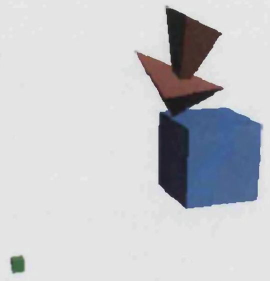


Figure 4.8: A position of the nomad on the path between the start and final positions that does not cause it to overlap with the obstacle.

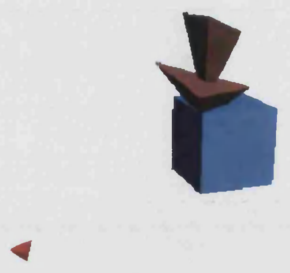


Figure 4.9: A position of the nomad on the straight hyperline between the start and final positions. The nomad is in contact with the obstacle, shown by the collision flag as a red tetrahedron.

4.3.3 Connected component labelling of binary trees

I have adapted Samet's algorithm for labelling connected components in two-dimensional quadtrees [64] to work with multi-dimensional binary trees, and implemented it to work on three-dimensional binary trees such as those created by dividing a sVLIs set-theoretically defined model.

The main traditional application area for Samet's algorithm is monochrome image processing, in which all leaf nodes of the quadtree will be either solid (black) or air (white). Samet also uses the term 'grey' nodes to denote non-leaf nodes which contain both black and white nodes; the equivalent of this in sVLIs is a non-leaf surface sub-model.

Samet's algorithm works as follows. A quadtree is traversed in postorder, visiting children in the order Northwest, Northeast, Southwest, Southeast - an example of this traversal is shown in figure 4.10. When a solid leaf node a is visited, its Eastern and Southern neighbours of equal or greater size are visited. If a has adjacent solid nodes, they are given the same label as a, generating a new label

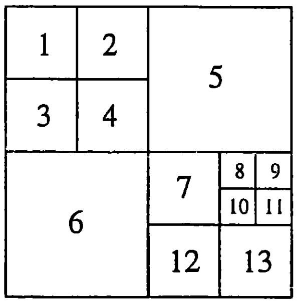


Figure 4.10: The order in which quadtree nodes are visited for Samet's algorithm.

if neither a nor its adjacent nodes are labelled. If both a and an adjacent solid node are already labelled, their labels are added to a label equivalence list. The label equivalences are merged after the tree has been traversed, and the elements in the tree are given their final labels which can be counted to find the number of connected components, or interrogated to determine which parts of the model are connected.

As with quadtrees, divided sVLIs models contain solid and air leaf sub-models, and also contain surface leaf sub-models. These latter are represented in my adaptation of the algorithm as either solid or air (selectable by the user), which respectively biases towards an underestimate or an overestimate of the number of disconnected components. Extension of the algorithm to handle simple surface sub-models analytically removes a large amount of uncertainty and means that the initial model need not be divided so finely, thereby creating a smaller binary tree to search - this is discussed further in section 4.3.5.

The general structure of Samet's algorithm has been retained in this adaptation for multi-dimensional binary trees, with some changes necessary to the functions that he suggests. First the model is divided. A recursive function to label the model is then applied, which traverses the model's binary tree in postorder; for a leaf model P, each of the neighbouring models Q_i in each dimension i of the

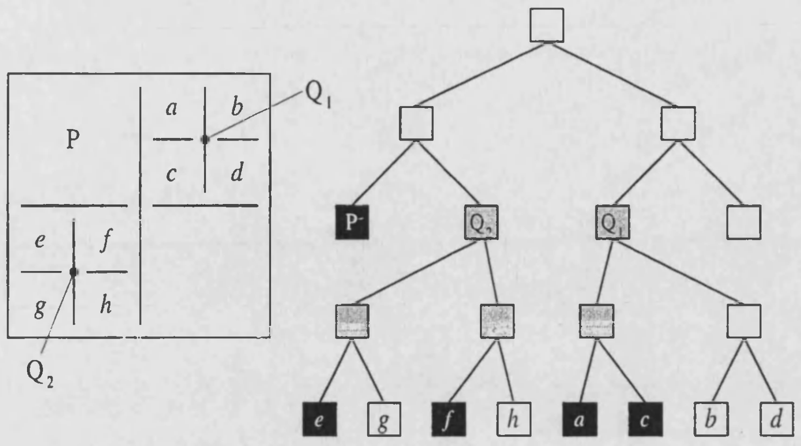


Figure 4.11: Example for binary tree connected-component-labelling algorithm.

n-dimensional space is found and is labelled as being adjacent to the leaf model. When labelling two models as adjacent, P is guaranteed to be a leaf model but Q_i may not be. If Q_i is not a leaf model then all of its children that are closest to P in the direction of the cut between P and Q_i must be recursively labelled. Figure 4.11 shows an example two-dimensional model where neither Q_1 nor Q_2 are leaf models. The binary tree for the model is also shown, with the children of Q_1 and Q_2 closest to P in the cut direction denoted by shaded boxes; these will be given the same label as P. The leaf nodes that will be labelled are shown as black boxes.

Pseudo code for the new connected component labelling algorithm is given below. Functions denoted by italics are further explained below the pseudo code.

Main function

```
divide the model
label(model)
merge equivalent labels
```

```
**function:** *label (model)*
if model is not a leaf
  *label (model's first child)*
  *label (model's second child)*
else
  do for each axis
     Q = *neighbour in axis*
     if not on an edge
        *label as adjacent (Q, model, axis direction)*
     end if
  end do
end if
```

```
**function:** *label as adjacent (model 1, model 2, axis direction)*
if model 1 is not a leaf model
  *label as adjacent (model 1's first child, model 2)*
  if axis direction is different to model 1's cut direction
     *label as adjacent (model 1's second child, model 2)*
  end if
else if *are connected (model 1, model 2)*
  *label with same label (model 1, model 2)*
end if
```

The first step is to *divide the model,* which produces a model divided into sub-models each small enough to ensure that connections are not inadvertently made between two disconnected objects. The division method used is one that divides until either the volume of the sub-model's boxes is smaller than the original model's volume by a ratio dependent on the required accuracy, or the sub-model contains no surface of any object. The volume ratio must be chosen so that the smallest sub-model boxes are smaller than the smallest feature on the model; for example if the smallest feature is one tenth of the length of the model's box, then the division volume ratio must be 10^{-3}. The box-swell factor (see section 2.2.2 on page 13) for the division must be set to zero, so that the sub-model's boxes do not overlap.

The function to *merge equivalent labels* is not explained here as it is essentially the same as in Samet's code.

The main loop of the labelling part of the algorithm (*label*) was changed from Samet's code for two reasons; firstly to accommodate a three-dimensional structure rather than Samet's two-dimensional quadtree, and secondly as there are only two child trees in a binary tree that must be recursed into instead of the four child trees in a quadtree.

The function to find the neighbour of a node that is greater or equal in size to that node (*neighbour in axis*) was completely rewritten to use a 'top-down' neighbour method instead of Samet's 'bottom-up' method (*gtequal_ adj_ neighbor*). Samet's bottom-up method is unsuitable for binary trees because it relies on the position of each node in relation to its parent and the layout of each node is not fixed as it is in 2^n trees; for example in a 2^n tree the position of each node is known by examining which child it is: the southwest (minimum x and y) child of a quadtree is always at the bottom left corner of a node, whereas in a two-dimensional binary tree a child that is at the minimum direction in the x axis could be at the minimum or maximum y position, depending on the y position of its ancestor node that is split in the y direction. The proposed *neighbour in axis* function uses a 'top-down' geometric method to find the neighbouring node: the centroid of the neighbouring box is found by translating the centroid of the current box

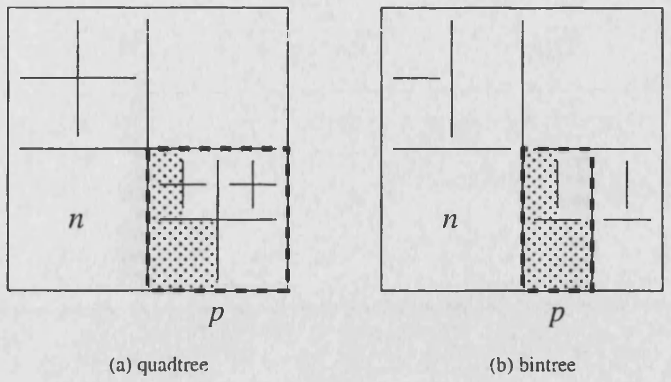


Figure 4.12: Labelling adjacent nodes of quadtrees and binary trees.

in the direction of the neighbouring box, and by a translational distance equal to the length of the side of the current box which faces in the direction of the neighbouring box. The box tree is then recursively traversed to find the largest box with that centroid, and the resultant box is returned as the neighbour. This method relies upon the box structure being divided with no overlap between the boxes, which is the reason for the box-swell factor being set to zero in the initial model division function.

The function *label as adjacent* labels all children of a node p that are adjacent to another node n with the same label as n. It was changed from Samet's *label_adjacent,* because identification of the children of a tree node that lie on the correct 'side' needs a different function to that used for quadtrees. The original function starts from a given node p, which is not necessarily a leaf node, and labels all children of p that are adjacent to a node on the Northern (Western) side of p with the same label as n. In order to do this the Northwest and Northeast (Northwest and Southwest) children are recursed into, labelling the Northwest and Northeast (Northwest and Southwest) solid leaf nodes with the same label as n. Figure 4.12(a) shows the leaf nodes adjacent to n that would be labelled by this function as shaded nodes, and the starting node p with a thick chain outline.

In my binary tree implementation of this function, the children of node p that are adjacent to node n are found by walking over p's tree recursively, and as children of p that have been divided in the same direction as the division between p and n are entered, only walking into the child that is in the direction of p. For example, using the two-dimensional binary tree of figure 4.12(b), p and n are divided in the x-axis. The first level of recursion will walk into both children of p since they are divided in the y-axis, and the next level of recursion will only select the sub-model with the least value in the x-axis, since that is in the direction of n. For this example the recursion would then be complete.

The function *are connected (model 1, model 2)* determines whether model 1 and model 2 are connected by evaluating the contents of each model in terms of air, surface or solid. Surface models are considered to be either solid or air, depending upon a global setting selected for each execution of the algorithm. This allows conservativeness in diagnosing connectivity, or in diagnosing disconnectedness. If both models are solid (or are considered to be solid) then they are connected; otherwise one of the models is air (or is considered to be air), and they are not connected.

The function to label two models with the same label (*label with same label*) is not explained here, as again it is essentially unchanged from Samet's original.

4.3.4 Results

The connected component algorithm has been implemented for the three-dimensional binary trees that are created by dividing a sVLIs set-theoretically defined model, and figures 4.13 to 4.18 show the results of running the algorithm on several such sVLIs models. The algorithm has automatically allocated each

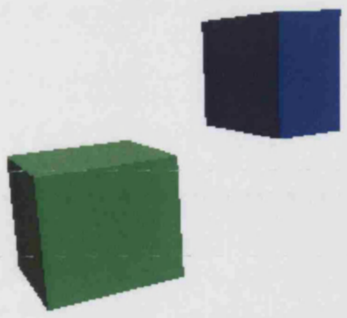


Figure 4.13: Connected components found for 'Two boxes' model.

separate connected component found within the models a different colour. The mode of the algorithm selected for these tests was that in which surface leaf boxes are considered to be solid.

Results for these particular models are summarised in table 4.1. The connected components of each model were labelled and counted at several different division resolutions - the value given in the table is the ratio of the volume of the smallest divided sub-model's box to the volume of the original model's box. For each of these resolutions the number of components found is listed. The algorithm was run on each model to compare the effect of the choice of modes, i.e. either where the surface leaf boxes are treated as solid boxes, or treated as air boxes. The 'Working resolution' column shows for each model the division resolution at (and below) which the algorithm consistently[1] labels the components correctly.

The most significant aspect of the results is that when surface boxes are considered to be air, the number of components returned does not behave in a pre-

[1]Although the number of components is correctly found for the racing car model at division resolution of 10^{-4}, this is due to fortunate alignment of the divided model's sub-models so that there are three areas of solid sub-models with air between them. A division resolution of 10^{-7} or finer is required to correctly separate the solid sub-models into three areas.

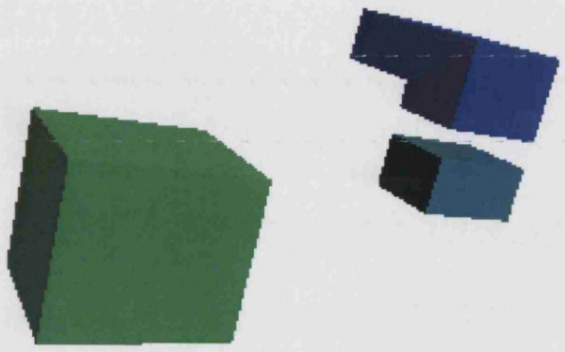


Figure 4.14: Connected components found for ‘Two L shapes and block’ model.

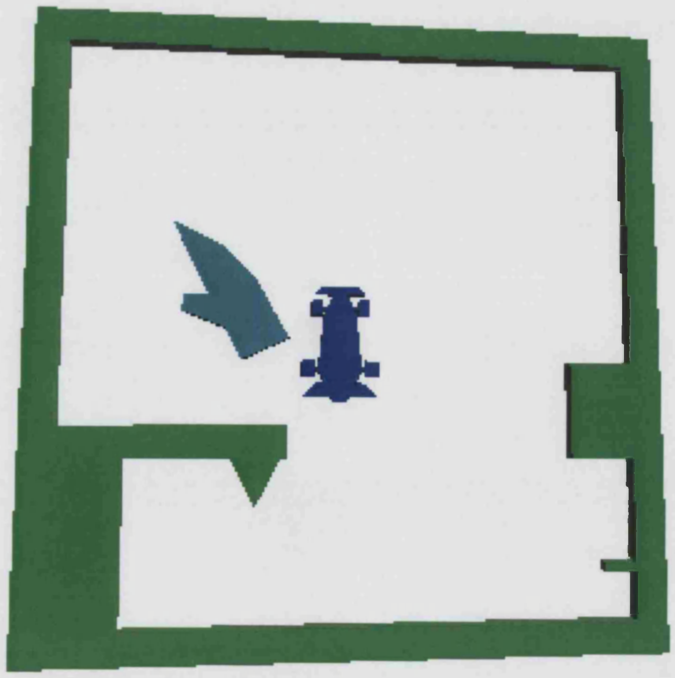


Figure 4.15: Connected components found for ‘Racing car’ model.

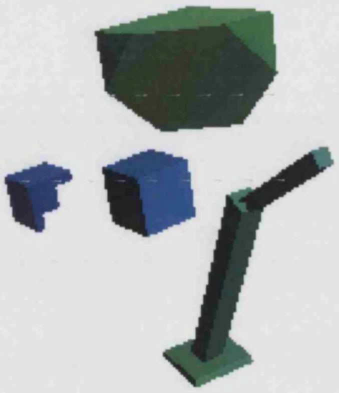


Figure 4.16: Connected components found for ‘Robot with obstacles’ model.

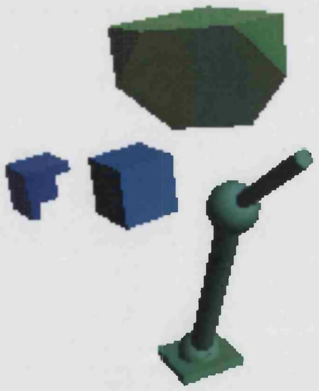


Figure 4.17: Connected components found for ‘Non-polyhedral robot’ model.

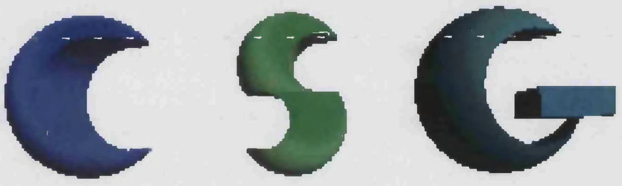


Figure 4.18: Connected components found for 'CSG' model.

Model	Compo- nents	Surface box type	Working resolution	Components found				
				10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}
Two boxes	2	solid	10^{-4}	1	2	2	2	2
		air	10^{-3}	2	2	2	2	2
Two L shapes and block	3	solid	10^{-5}	1	2	3	3	3
		air	10^{-4}	2	3	3	3	3
Racing car	3	solid	10^{-7}	1	3	4	4	3
		air	-	0	0	0	9	4
Robot with obstacles	4	solid	10^{-4}	3	4	4	4	4
		air	10^{-7}	0	0	1	11	4
Non-polyhedral robot	4	solid	10^{-4}	3	4	4	4	4
		air	10^{-7}	0	1	5	11	4
CSG	3	solid	10^{-5}	1	4	3	3	3
		air	-	0	4	5	4	5

Table 4.1: Results of the binary tree connected component algorithm. The number of components found is shown at several division resolutions and for the algorithm considering surface leaf sub-models to be either solid or air.

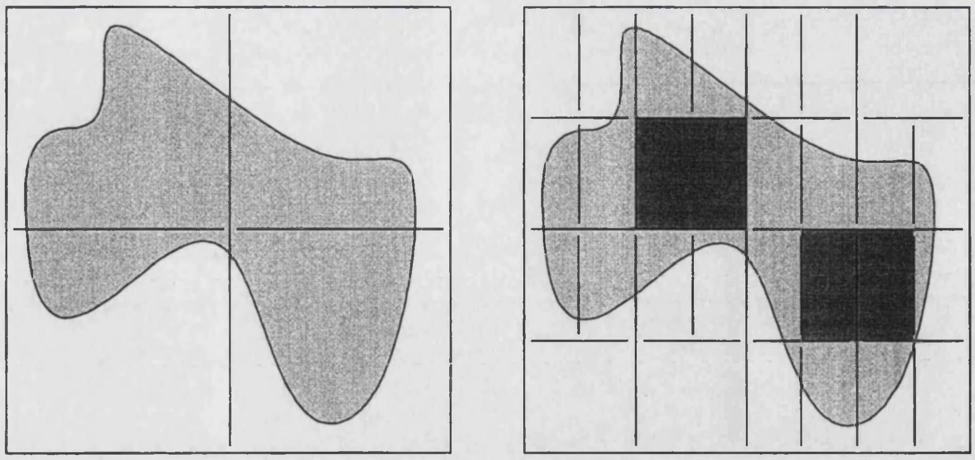


Figure 4.19: Failure situations for the connected component algorithm when surface boxes are considered to be air boxes.

dictable manner as the resolution of division is made finer. Generally the number of components will be too few with coarser division resolutions, as exemplified in figure 4.19(a), owing to the boxes being too large to contain just solid parts of the components. This results in the model only consisting of surface boxes that are considered to be air, so no component is found. The number of components increases to the correct value as division resolution is made finer, however for some models (i.e. 'Racing car', 'Robot with obstacles', 'CSG') the number of components goes above the correct number as division is made finer. This error occurs because the smallest boxes that the model is divided into contains only parts of the components within solid boxes, and these solid boxes are not connected, even though in reality they are part of the same connected component; this is shown in figure 4.19(b). As division is made even finer, the number of components found should fall to the true value.

When surface boxes are treated as being solid boxes by the algorithm, the number of components found is generally an underestimate of the true number of components at coarser division resolutions, increasing to the correct number as the division is made finer. The exception with the example models used is the 'CSG' model which erroneously rises to four components at a division resolution of 10^{-4} because the open area of the 'C' shape is incorrectly identified as containing a component. This is due to the method used to evaluate the contents of the model within sVLIs which is not exact for non-polyhedral models or for polyhedral models with contents greater than three. To produce the correct answer the model must be divided more finely. This precludes the use of an automatic technique for finding the optimal division-resolution, where resolution is made finer until the number of components no longer increases. To ensure correct component labelling the division resolution must be made fine enough so that no model features can be contained in the smallest division box.

4.3.5 Dealing with surface sub-models analytically

For the binary tree connected component labelling algorithm described previously, the models used are divided into solid, air and surface sub-models. The surface sub-models are considered to be either air or solid, which can give under- or over-estimates respectively of the number of separate components. If the connectivity between surface sub-models could be found analytically, then the number of components could be calculated exactly, and the accuracy of the calculation would not rely upon the resolution of the initial model division.

Initially only polyhedral sub-models have been considered, as they are simpler to deal with than general models, and can be used to approximate many non-polyhedral objects. Two properties of a sub-model need to be computed: the number of disconnected components in the sub-model and the connectivity between a surface sub-model and its neighbouring sub-models.

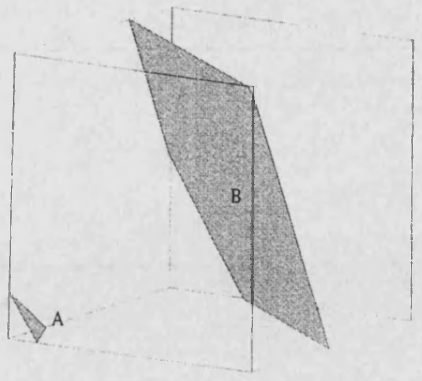


Figure 4.20: A sub-model containing two planar half-spaces that do not cross within the sub-model's box.

If the model can be divided such that the number of disconnected components in each sub-model can be guaranteed to be zero or one then the number of connected components in the entire model can be ascertained by examining the connectivity between a surface sub-model and its neighbouring sub-models. In order to see how a model may be divided into sub-models that contain at most one component it is necessary to examine how sub-models containing any number of planar half-spaces (the most primitive entity for the model) are dealt with in the division process. If a sub-model contains more than three planar half-spaces it is divided. If a sub-model is of the smallest size allowed by the division resolution, then the model contains features smaller than the division resolution, and the connectivity between these features cannot be computed. For other cases, any given sub-model will contain either one, two or three planar half-spaces. Each of these possibilities is now considered in turn.

One planar half-space: A sub-model containing one planar half-space can only contain one component.

Two planar half-spaces: If a sub-model contains two planar half-spaces that cross within the sub-model's box then the sub-model must consist of only

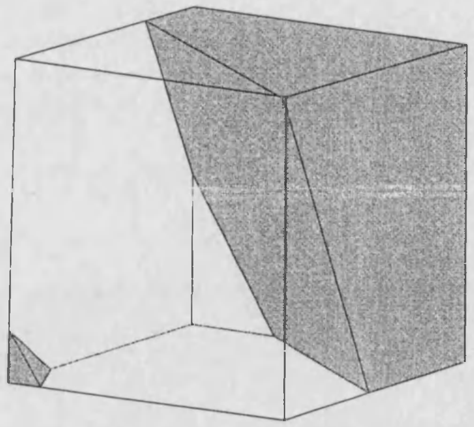


Figure 4.21: Two planar half-spaces that do not intersect in the sub-model's box, and result in two unconnected components.

one component. If on the other hand a sub-model contains two planar half-spaces that do not cross (for example A and B in figure 4.20), then the sub-model can contain two components if the half-spaces normals are facing towards each other and they are combined with the union operator (such as $A \cup B$, shown in figure 4.21). Therefore to ensure that no sub-model containing two half-spaces can contain more than one component any sub-models containing two half-spaces should be divided if the half spaces do not cross within the sub-model's box.

Three planar half-spaces: If the three half-spaces cross at a point within the sub-model's box then the sub-model can contain only one disconnected component as the half-spaces are connected through the point at which they cross. An example of this is shown in figure 4.22. If they do not cross at a point then they could contain one component (for example the situation shown in figure 4.23) or more than one component (for example figure 4.24). Therefore to ensure that sub-models which contain three half-spaces can only contain one component, sub-model boxes which contain three planar half-spaces must be divided if the half-spaces do not cross at a point within the sub-model's box.

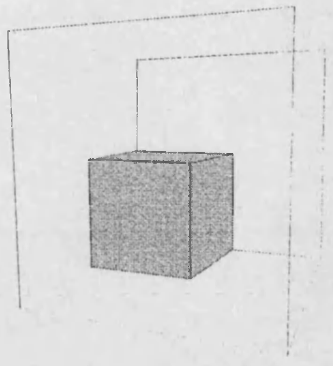


Figure 4.22: Three planar half-spaces that intersect at a point, resulting in one component.

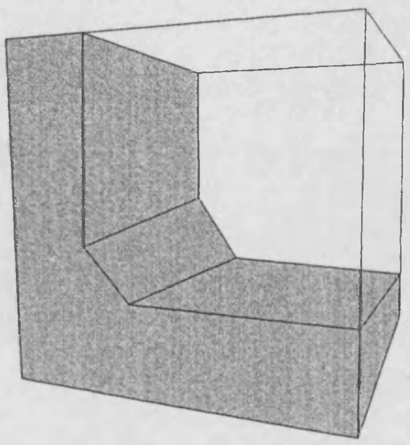


Figure 4.23: Three planar half-spaces that do not intersect at a point and result in one component.

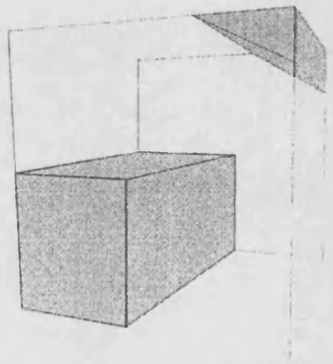


Figure 4.24: Three planar half-spaces that do not intersect at a point and result in several unconnected components.

As the number of components within a sub-model has been constrained to be at most one, the connectivity between a surface sub-model and its neighbouring sub-model must be calculated. In order to do this, let R be the rectangle of contact between the two sub-models' boxes. If there exists a point in R which membership tests as solid for both sub-models, then the two sub-models are connected. In general, finding whether such a point exists is not straightforward, but if both of the sub-models consist only of planar sets then the following properties can be used to test if the sub-models are connected:

- If they each contain only one plane, and it is the same plane, then they are connected.
- In other cases the connection between the sub-models must be within the rectangle R. Points at the intersection of the planes with R and the intersection points of the planes must be tested, and if any point is solid or surface with respect to the model, then the sub-models are connected. Otherwise if all points are air, then the sub-models are not connected. The connectivity of a sub-model to another sub-model only needs to be tested at points where the edges of the rectangle of overlap intersect the planes or at points where two planes intersect each other because the connectivity of a sub-model can only change at the surface of the sub-model.

From this, the following method has been devised for connected component labelling of set-theoretically defined polyhedral models. Based on the algorithm described in section 4.3.3 for connected component labelling of binary trees, the functions *divide the model* and *are connected (model 1, model 2)* have been modified as described below.

The new *divide the model* will divide the model until the smallest sub-models satisfy one of the following conditions:

- the sub-model contains only one planar half-space.
- the sub-model contains two planar half-spaces and these cross to produce a line within the sub-model's box.
- the sub-model contains three planar half-spaces and these intersect at a point within the sub-model's box.

The changed function *are connected* determines whether two sub-models are connected by using several steps, which increase in computational complexity. As soon as a step determines that the sub-models are connected, the function can finish. The steps are:

1. Test if both models are solid. If they are, then they are connected.
2. Test if both models contain only one planar primitive. If they do, and it is the same primitive for both models, then they are connected.
3. Test if both models are polyhedral. If they are, and they contain three or less primitives each, then find the points of intersection (if any) between each plane in model 1 and each plane in model 2 within the rectangle of overlap R. Also find the points where the planes of the models intersect the edges of R. If any of these points are solid then the sub-models are connected; otherwise all of the points are air which means that the sub-models are disconnected.
4. If step 3 fails, i.e. the models are not polyhedral or they contain more than three primitives (in which case the feature size of the model is smaller than the smallest division resolution) then the analytical solution cannot work.

Model	Compo-nents	Surface box type	Working resolution	Components found				
				10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}
Two boxes	2	solid	10^{-3}	2	2	2	2	2
		air	10^{-3}	2	2	2	2	2
Two L shapes and block	3	solid	10^{-5}	1	3†	3	3	3
		air	10^{-4}	2	3	3	3	3
Racing car	3	solid	10^{-6}	1	2	3	3	3
		air	10^{-6}	11	12	10	3	3
Robot with obstacles	4	solid	10^{-4}	3	4	4	4	4
		air	10^{-6}	4†	7	6	4	4
Non-polyhedral robot	4	solid	10^{-5}	3	3	4	4	4
		air	10^{-7}	2	5	6	11	4
CSG	3	solid	10^{-5}	1	4	3	3	3
		air	-	0	5	5	6	5

Table 4.2: Results for the connected component labelling algorithm that handles polyhedral surface models analytically.
†The number of components found is correct but the components are labelled incorrectly.

A trade-off has been made between the amount of analytical work that is done for a sub-model and the amount of division that will be done. The surface boxes are considered to be either air or solid, globally selectable by the user, as for the non-analytical solution. If both models are solid, or they are surface and surface is considered to be solid, then the models are connected.

If none of these steps determines that the models are connected then *are connected* terminates with the conclusion that the models are disconnected.

4.3.6 Results

The algorithm was tested with the same models as the binary tree connected component algorithm from section 4.3.3, giving the results shown in table 4.2.

When surface boxes are considered to be solid, the number of components is calculated exactly for all of the polyhedral models (i.e. all of the models except the 'CSG' and 'Non-polyhedral robot' models) at a coarser division resolution than the division resolution required for the non-analytical connected component labelling algorithm to give the correct results. The benefit of using a coarser division resolution is that the components can be labelled more quickly. For the non-polyhedral models there is no change for the 'CSG' model, however, for the 'Non-polyhedral robot' model, the number of components was found correctly at a coarser division by the non-analytical method. This is because the model contains both polyhedral and non-polyhedral parts and at the division resolution of 10^{-4}, the polyhedral parts of the model are not divided as finely as when using the non-analytical method and one of the polyhedral sub-model's boxes is adjacent to a non-polyhedral sub-model's box so the two are considered to be connected. To resolve this situation, a division resolution finer than the smallest feature of the model should be used.

When surface boxes are considered to be air, the number of components is found correctly at the same division resolution or at a coarser division resolution (for the 'Robot with obstacles' model) than that achieved by the binary tree labelling method where surface boxes are not handled analytically. Additionally, the correct number of components is found for the 'Racing car' model for which the non-analytical method could not find the correct result. The number of connected components could again not be found for the 'CSG' model, which is non-polyhedral so no improvement was expected for the analytical method over the non-analytical method.

4.4 Future work

If the point connectivity method suggested is applied to a model that consists of two components that are disconnected but are not separated by an infinite plane it will not be able to compute whether the components are connected or not. It may be possible to use a hybrid scheme that finds gaps between components by recursively spatially dividing the model to find sub-models that are completely air, and finds connections by using the point-connection line segment intersection method. An alternative approach would be to use the point connectivity method within the binary tree connected component labelling method to answer the question of whether two surface sub-models are connected.

The connected component algorithm has been designed so that it is not restricted to work in a certain number of dimensions. A multi-dimensional version of the algorithm could be implemented using the sVLIs ^{m} multi-dimensional set-theoretic modeller [12].

Extension of the analytical method for connected component labelling to non-polyhedral models should result in more accurate results at coarser model division resolutions for these types of model. It should be relatively straightforward to calculate exact connectivity information for spheres and cylinders, so these would provide an ideal starting point for such work.

It may be beneficial to investigate whether adapting a different connected component labelling algorithm for use with binary trees is more efficient than the adapted Samet's algorithm. Samet and Tamminen's 1988 work [67] would be interesting to investigate since that was designed for use with n-dimensional binary trees. It is however more complicated to implement since it uses an active border to store information about whether scanned elements at the edge of the scanned region are air or solid.

An algorithm that combines the proposed binary tree algorithm and the analytical binary tree algorithm for polyhedral models would give accurate connectivity information on complicated set-theoretically defined models. For simple polyhedral parts of the model, the analytical approach would be used and for more complicated parts the algorithm would revert to using the binary tree approach. The initial model division would not divide sub-models which are simple enough for their connectivity information to be ascertained analytically. Alternatively it may be possible for this *hybrid* algorithm to be built around adaptive model sub-division, so that a model would be divided as the connectivity information is being computed, and only areas of the model which could not be solved at the current division resolution would need to be divided more finely.

4.5 Conclusions

A method of testing whether or not two points in a set-theoretically defined model are connected has been introduced. This uses recursive techniques that compute the intersection of line segments with the model in an attempt to find a connected path between the two points that lies completely within air or solid parts of the model. Two variants of this method are proposed, the first of which has been implemented in a multi-dimensional set-theoretic modeller to find a path through the configuration space of a nomad model that translates and rotates around an obstacle model. The second variant is similar and has therefore not been implemented.

A connected component labelling method based on the adaptation and extension of an existing quadtree algorithm by Samet has been presented and successfully applied to several models. This method operates by firstly dividing a model to create a binary tree, and then labelling the connected components in the binary tree, thus labelling the connected components of the model.

Chapter 5

Minimum distance

5.1 Introduction

Minimum distance information for solid geometric models has many applications, including path planning, configuration-space map generation and component analysis [56, 69]. As Cameron has stated, "minimum distance is far and away the most important factor for robot navigation and choreography" [14]. In the field of robotics, advanced path-planning systems, such as the one proposed by Paden *et al* in 1989 [56], need distance information. Many configuration-space (Lozano Pérez [50]) algorithms also need to perform distance computations: an example is Siméon's algorithm [69]. As far as I have been able to ascertain, there has been no work carried out to determine the minimum distance between two set-theoretically defined solid models, i.e. CSG or boolean solid models, although Cameron [13] has examined the intersection query for CSG models which tests for touching objects, i.e. those with a minimum distance of zero. An efficient algorithm for computing distances between set-theoretically defined models would make the implementation of some of the path-planning systems mentioned above possible for such models. Distance information on engineering components that have been modelled using set-theoretic geometry could be used to check the structure of components (for example, the cylinders in an engine block must be more than a certain distance apart otherwise there will be insufficient space for coolant flow between them). The problems that need to be solved are those created by the fact that set-theoretic models are stored in a non-evaluated format.

Existing approaches to finding the minimum distance between models stored using a variety of representation methods are detailed in section 5.2. Most of these find only the minimum distance between convex models and not concave models, as, in general, faster solutions to the minimum distance problem can be found for convex models. Methods such as Red's [61] to find the distance between non-convex polyhedra rely on finding the distances between the convex components that have been combined to create the non-convex polyhedra. It is not always possible, however, to decompose non-convex *curved* objects into unions of convex objects: a block with a cylindrical hole cut through it is the classic example. The convex decompositions of some models would need a large number of convex components to represent them, for example a prosthetic socket that follows the contours of a human limb has many indentations, so the convex decomposition of this would be complicated.

The *minimal translational distance* [15] is a metric that is closely related to minimum distance, and is defined as "the length of the shortest relative translation that results in two objects being in contact". It is equal to the minimum distance between two non-touching objects, but for intersecting objects it is a measure of penetration. *Minimum distance in a fixed direction* is another special case of minimum translational distance. It is a useful measure for some applications, mainly collision detection and avoidance. If two models are known to be moving

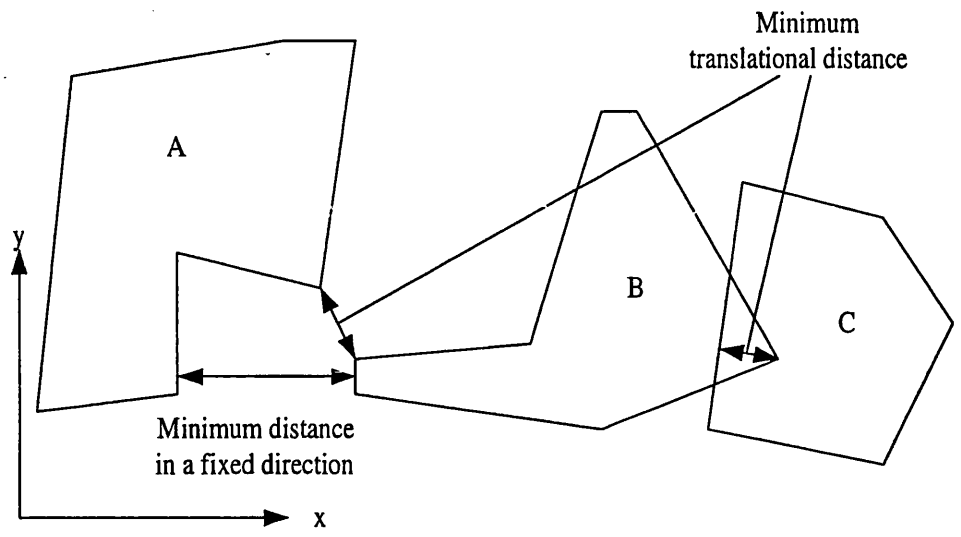


Figure 5.1: Minimum translational distance and minimum distance in a fixed direction

towards each other in a fixed direction, then the minimum distance in that fixed direction will give the maximum translation of the models that can occur before collision. Figure 5.1 shows three example polygons A, B and C, the minimum translational distance between A and B and B and C, and the minimum distance in the fixed direction parallel to the x axis between A and B.

A special case of minimum-distance calculation is *incremental distance calculation* or *the tracking problem,* which arises when two objects are moving, and the minimum distance between objects is required to be calculated as the objects move. The minimum distance at the previous position of the objects can be used to find the distance for the present position more efficiently [48].

I have designed and implemented a method for finding the minimum distance between general non-convex or convex set-theoretic models including polyhedra and models containing curved surfaces. This method uses adaptive recursive spatial division of a set-theoretic model and pruning of the resultant search space by discarding widely separated sub-models. This technique is explained in detail in section 5.3.3.

5.2 Literature survey

Work on finding the minimum distance between two objects has been done on several types of objects including point sets [46], polygons [68, 19], convex objects in three or more dimensions [29, 37, 6, 28, 82, 74], and non-convex polyhedra [61].

5.2.1 Point sets

Lee and Preparata (1984) describe the problem of finding the two closest points in a set of points in their survey of computational geometry [46]. The solution to this is generally a search of all the points, although the divide-and-conquer technique proposed by **Bentley and Shamos (1976)** [4] finds the minimum distance more efficiently in $O(n\, log\, n)$ time.

5.2.2 Polygons

Schwartz (1981) gives an algorithm [68] for computing the minimum distance between two convex two-dimensional polygons P_1 and P_2. He utilizes the fact that the minimum distance between P_1 and P_2 is equivalent to the distance between the origin and the convex set that is the *Minkowski difference* of polygons P_1 and P_2 ($P_1 \ominus P_2$). The Minkowski difference is defined as the Minkowski sum of P_1 and the reflection in the origin of P_2, i.e. $P_1 \oplus (-P_2)$ where $P_1 \oplus P_2$ is the Minkowski sum of P_1 and P_2. Wise [77] has given a set-theoretic implementation of the Minkowski sum for convex polyhedra. Schwartz gives an $O(\log^2 n)$ procedure for finding $P_1 \oplus P_2$, where n is the total number of vertices in P_1 and P_2. The overall algorithm for determining minimum distance also works in $O(\log^2 n)$ time.

Chin and Wang (1983) give an $O(\log n)$ algorithm for finding the minimum distance between two two-dimensional polygons [19], one of which must be convex. The non-convex polygon must be a *simple* polygon, i.e. a polygon whose edges do not intersect themselves and whose vertices are represented by ordered pairs, in either Cartesian or Polar coordinates. The time complexity of the algorithm is $O(n+m)$, where n is the number of points in the convex polygon and m is the number of points in the simple non-convex polygon. For efficiency Chin and Wang's algorithm uses the *visible edge chains* of the polygons, and they prove that the minimum distance between a convex polygon and a non-convex (or convex) polygon is the same as the distance between the *visible edge chains* of the polygons. A point q in a polygon Q is said to be *visible* from a point o if the line segment (o, q) does not intersect with any other point on the boundary of Q. A *visible edge chain* $V(Q, o)$ is a sequence of edges of Q that are all visible from o. Figure 5.2 shows the visible edge chains $V(Q, o_1)$ and $V(Q, o_2)$ for two points o_1 and o_2 of the polygon S. The *visible chain* $V(Q, S)$ of a polygon Q from a polygon S is defined by

$$V(Q,S) = \bigcap_{x \in S} V(Q,x)$$

i.e. it is the overlap of all the visible edge chains for all points in S. For the example polygons in figure 5.2 it is the same as $V(Q, o_1)$.

Chin and Wang's algorithm starts by finding the visible chain V of the non-convex polygon (Q) from the convex polygon (P), which can be performed in $O(m)$ time, where m is the number of vertices in Q. A scan is performed, starting at the first vertex v in V and the vertex p in Q that is closest to v. The scan finds the minimum distance between v and the line segment in Q that precedes p or the

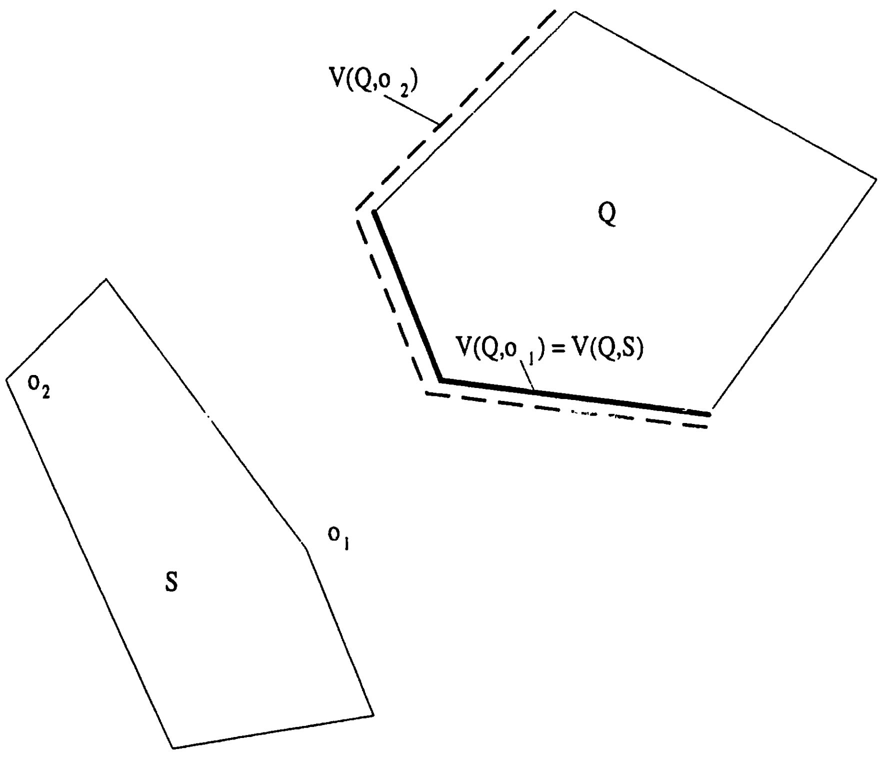


Figure 5.2: Visible chains for a polygon.

distance between p and the line segment in V that precedes v, deciding between these two possibilities on the basis of whether the closest point in Q to v precedes p or not. The scan also advances through either V or Q at each iteration of the algorithm on this same basis. The global minimum distance is the minimum of the minimum distances that are found at each iteration of the algorithm. The algorithm is efficient because each point in P need not be compared to each line segment in V and vice versa.

Chin and Wang also give an algorithm which finds the minimum distance between two convex polygons P_1 and P_2. To do this, the visible chains $V_1 = V(P_1, P_2)$ and $V_2 = V(P_2, P_1)$ are computed. The distance between V_1 and V_2 can then be quickly found using a binary search. For any point in V_1, the distance from V_2 increases in one direction along V_1 and decreases in the other direction along V_2. Starting at the mid-point r_0 of one of the visible chains V_i, the direction in which the distance to the other visible chain V_j decreases is determined. The mid-point between r_0 and the end of V_i is then found and this binary division of the chain is continued recursively. The search is also performed on the other visible chain V_j.

5.2.3 Convex objects in three or more dimensions

Gilbert, Johnson and Keerthi (1988) [29] give an algorithm for convex polytopes and their spherical extensions, where a spherical extension of radius r to a polytope consists of all the points that are within a distance of r from the polytope and all the points in the polytope - it is equivalent to the Minkowski sum of the polytope and a sphere of radius r; for example in three dimensions the spherical extension of a point would be a sphere, and the spherical extension of a line would be a cylinder with hemi-spherical ends. The approach is similar to that taken by Schwartz [68] for two-dimensional objects in that it transforms the minimum distance problem into one of finding the Minkowski difference of the two sets of objects and then finding the minimum distance of this from the origin. However the algorithm given by Gilbert, Johnson and Keerthi has the advantage that it generalizes to any number of dimensions. The Minkowski difference does not actually have be evaluated, as distances and *support functions* (see below) can be computed directly from the two sets. Gilbert, Johnson and Keerthi give experimental results to show that the overall computational effort of the algorithm is approximately linear with respect to the total number of vertices specifying the two objects.

Gilbert, Johnson and Keerthi's algorithm for finding the minimum distance between sets K_1 and K_2 is summarised here. Let K be the Minkowski difference of K_1 and K_2 i.e. $K = K_1 \ominus K_2$. Perform the following steps:

1. Set $k = 0$ and initialize the set V_0 to be a subset of K containing at least 1 element and less than $m + 2$ elements where m is the dimensionality of K.
2. Find the point v_k which lies within V_k and is nearest to the origin.
3. Find the point q lying within K which has the largest projection onto $-v_k$. If the projection is of equal length to the distance between the origin and v_k but in the opposite direction, then v_k is the closest point in K to the origin; terminate the search as the algorithm has determined the distance between K_1 and K_2 to be the distance between the origin and v_k.
4. Let $\hat{V}_k$ be the smallest subset of V_k within which v_k lies. Set $V_{k+1} = \hat{V}_k \cup q$. Increment k. Continue from step 2.

A simple two dimensional example can be used to demonstrate the algorithm. Figure 5.3 illustrates four polygons: K_1, K_2, $-K_2$ (the reflection of K_2 in the origin) and the Minkowski difference $K = K_1 \oplus -K_2 = K_1 \ominus K_2$. Figure 5.4 shows the quantities used in the algorithm for the point set K made up of points $\{s_0, \ldots, s_5\}$ after V_0 has been initialized to $\{s_0, s_1, s_2\}$ and v_0 has been found. The point in K with the largest projection onto $-v_0$ has been found as s_4, and q has been set to this. The projection has a different length to v_k, so step 4 is performed and $\hat{V}_0$ is set to the smallest subset containing v_0 which is $\{s_0, s_2\}$, so $V_1 = \{s_0, s_2, s_4\}$. In the next iteration of the algorithm, v_1 will be set to s_4, and the algorithm will terminate, having found the distance between K_1 and K_2 as the distance between the origin and s_4.

The algorithm needs to be able to find the point q which lies within K and which has the largest projection onto a vector v, as used in step 3. This is done by using K_1 and K_2, and does not require K ($= K_1 \ominus K_2$) to be evaluated. Given a convex set X, and that $x \cdot y$ denotes the inner product of x and y, the support

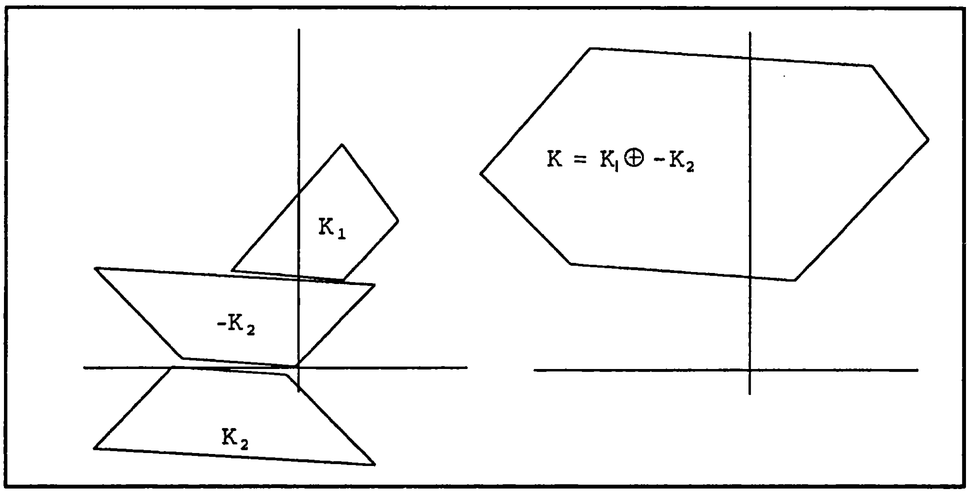


Figure 5.3: Two example polygons and their Minkowski difference.

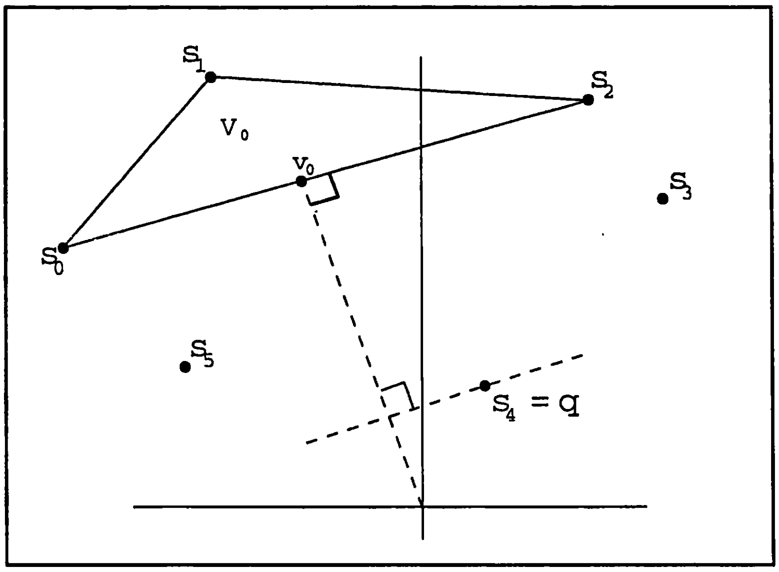


Figure 5.4: Example for GJK's algorithm.

function of X is defined by:

$$h_X(\eta) = max\{x \cdot \eta : x \subset X\}$$

$s_X(\eta)$ denotes any solution of the above, i.e.:

$$h_X(\eta) = s_X(\eta) \cdot \eta, s_X(\eta) \subset X$$

Informally, $s_X(\eta)$ is the point in X which is furthest from the origin in the direction of the vector η, so in figure 5.4, $s_X(-p)$ is the point s_4. In step 3 of the algorithm described earlier, point q is found as the point $s_X(-v_k)$, and the length of the projection is given by $h_X(-v_k)$.

Gilbert, Johnson and Keerthi state that to calculate $h_K(\eta)$ and $s_K(\eta)$ [where $K = K_1 \ominus K_2$] without evaluating K it is possible to use:

$$h_K(\eta) = h_{K1}(\eta) + h_{K2}(-\eta)$$

and

$$s_K(\eta) = s_{K1}(\eta) - s_{K2}(-\eta)$$

This allows the support function for the set K to be computed directly from K_1 and K_2 without having to find the Minkowski difference of the two sets.

Hurteau and Stewart (1988) [37] propose a minimum-distance algorithm for objects represented using constructive solid geometry, but the only primitives that their algorithm can deal with are convex polyhedra and finite cylinders, and the only operator that can be used to combine these primitives is the union operator. Clearly the variety of objects that can thus be represented is severely restricted, and although this may not be a great problem for some robotics applications, it is not sufficiently versatile as a general engineering component modelling tool.

Bobrow (1989) [6] describes a minimizing technique for finding the minimum distance for convex polyhedra and their spherical extensions. The polyhedra are considered to be made up of the intersection of several planar half-spaces, and the algorithm works directly on the points and normals that define these half-spaces, so bounding lines and vertices do not need to be explicitly calculated.

Bobrow's algorithm starts with two points p_1 and p_2, one inside each object. (Points near to the centroids of the objects are good initial points.) Two points, x_1 and x_2, which are on the surfaces of the objects and which lie on a line between p_1 and p_2 are found[1]. If x_1 and x_2 are not the closest points of the objects, then search directions s_1 and s_2 are determined from x_1 and x_2. s_1 is the projection of $v = x_1 - x_2$ onto the surface upon which x_1 lies, i.e. it is a vector along the surface of the object that contains p_1, in the direction in which x_2 lies from x_1. Similarly for s_2. The closest points x_1' and x_2' in the directions of s_1 and s_2 are found, and the algorithm is repeated using these points as x_1 and x_2, until the closest points between the two objects are found. Figure 5.5 illustrates the points and search directions found after one iteration of the algorithm.

Two parts of the algorithm deserve further explanation, as they are not trivial

[1]Bobrow also suggests an alternative of the two vertices with the smallest projections onto this line.

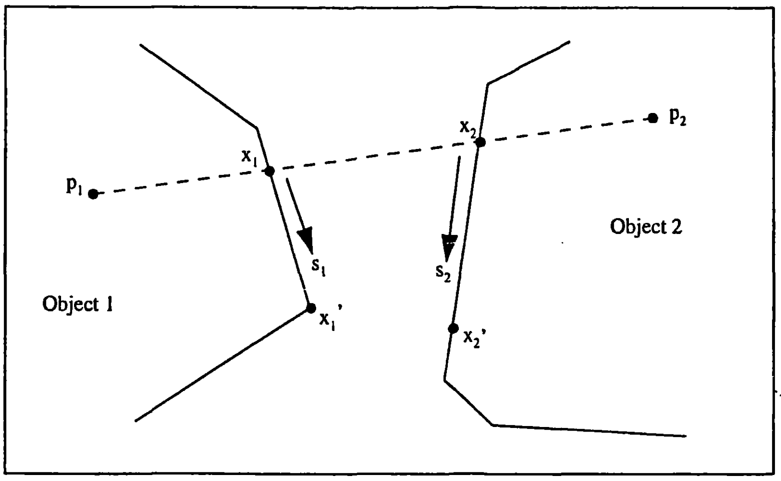


Figure 5.5: Points and search directions for example objects for Bobrow's algorithm.

and are essential to its operation. These parts are the check to see if x_1 and x_2 are the closest points, and the finding of the new search directions.

Bobrow uses *Kuhn-Tucker conditions* to determine if x_1 and x_2 are the closest points for the objects. If these conditions are satisfied then x_1 and x_2 are the closest points. The Kuhn-Tucker conditions are expressed by:

$$-v = \sum_{i=1}^{k_1} \alpha_{1,i} n_{1,i}$$

$$v = \sum_{j=1}^{k_2} \alpha_{2,j} n_{2,j}$$

where $v = x_1 - x_2$, k_1 is the number of planes that x_1 lies on, k_2 is the number of planes that x_2 lies on, $\alpha_{1,i}$ and $\alpha_{2,j}$ are positive scalar multipliers, $n_{1,i}$ are the normals of the planes that x_1 lies on and $n_{2,j}$ are the normals of the planes that x_2 lies on.

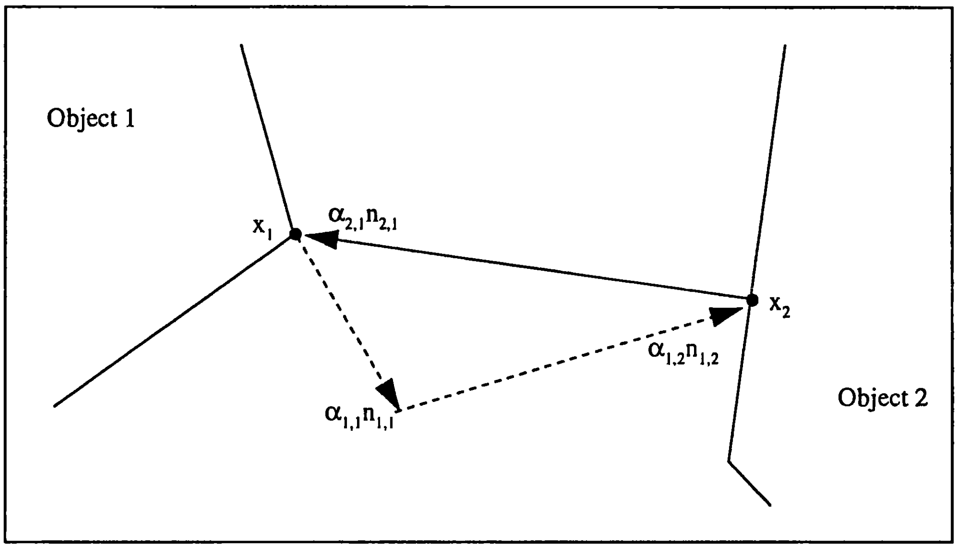


Figure 5.6: Example of variables used for evaluating the Kuhn-Tucker conditions in Bobrow's algorithm.

Figure 5.6 illustrates a 2-dimensional example. x_1 and x_2 have been set to the points x_1' and x_2' from figure 5.5. In this case, the Kuhn-Tucker conditions can be satisfied as shown by the lines in the figure. The dashed line shows that $-v = \alpha_{1,1}n_{1,1} + \alpha_{1,2}n_{1,2}$ and the solid line shows that $v = \alpha_2 n_2$ for $\alpha_{1,1}, \alpha_{1,2}, \alpha_2 > 0$.

Finding the new search directions uses the scalar multipliers $\alpha_{a,b}$ $(a = 1, 2; b = 1...k_a)$ found when attempting to satisfy the Kuhn-Tucker conditions. For each point x_1 and x_2, the plane with the largest value of $\alpha_{a,b}$ is found and v (or $-v$) is projected onto that plane to give the search direction.

An example will clarify Bobrow's method. The initial state is shown in figure 5.5, where x_1 and x_2 have been found by finding points on the surfaces of the objects that lie on the line between p_1 and p_2. The first step is to find out if x_1 and x_2 are the closest points on the objects. This is done by attempting to satisfy the Kuhn-Tucker conditions, which in this case are $x_2 - x_1 = \alpha_1 n_1$ and $x_1 - x_2 = \alpha_2 n_2$ as both points lie on one plane only. Whatever positive values of α_1 and α_2 are used, these conditions cannot be satisfied. Search directions are next found by projecting v and $-v$ onto the planes with the largest values for α. In this case,

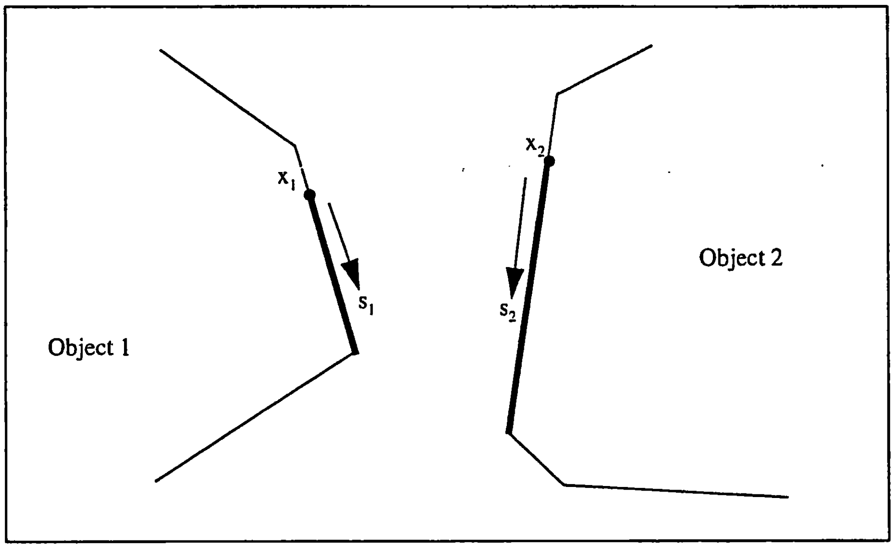


Figure 5.7: The line segments used to find new closest points for Bobrow's algorithm.

as both points lie on one plane each, then v and $-v$ are projected onto the planes on which the points lie. The search directions can be seen in figures 5.5 and 5.7 as s_1 and s_2. The line segments from x_1 in the direction of s_1 and from x_2 in the direction of s_2 are found. These are represented in figure 5.7 by the thick lines. The closest points between these two line segments are found, shown in figure 5.5 as x_1' and x_2'. Next the process is repeated using x_1' and x_2' instead of x_1 and x_2. The Kuhn-Tucker conditions are now $x_2 - x_1 = \alpha_{1,1}n_{1,1} + \alpha_{1,2}n_{1,2}$ and $x_1 - x_2 = \alpha_2 n_2$ where $\alpha_{1,1}, \alpha_{1,2}, \alpha_2 > 0$. These can be satisfied as shown by the lines in figure 5.5. The dotted line shows that the first condition can be satisfied, and the solid line shows the same for the second condition.

Because the Kuhn-Tucker conditions have been satisfied, then these points x_1' and x_2' must be the closest points on the two objects.

Gilbert and Foo (1990) [28] extend the work of Gilbert, Johnson and Keerthi in 1988 [29] to handle solids with curved surfaces as well as polyhedra. This is done by providing support functions for a sphere, an ellipsoid, a section of an ellipse and a section of a circle. Support functions for more complicated objects can be computed by combining the support functions of simple objects. General procedures for computing the support properties of objects created by taking the union or Minkowski sum of simple objects are given in terms of the support properties of the simple objects; however there is no general and simple procedure for computing the support properties of set intersections. The computational effort for the distance algorithm is the same as Gilbert, Johnson and Keerthi's original 1988 algorithm [29] , i.e. approximately linear with the total number of elements.

The algorithm by **Zeghloul and Rambeaud (1996)** [82] is similar to Bobrow's algorithm, however it solves the *'zig-zagging problem'*[2] and changes the test for whether the Kuhn-Tucker conditions are satisfied. Like Bobrow's, the algorithm works on convex polyhedra defined as the intersection of planes and is linear in the total number of planes. However many of the slowest cases for Bobrow's algorithm are made more efficient. Overall, Zeghloul and Rambeaud claim that their approach increases the speed of Bobrow's algorithm by a factor of more than two.

The major improvement is that the 'zig-zagging problem' identified in Zeghloul and Rambeaud's 1992 paper [81] is resolved. When the Kuhn-Tucker conditions are only satisfied on one object, Bobrow's algorithm will set the search direction for the object on which the Kuhn-Tucker conditions are satisfied to null. Figure 5.8 illustrates the problem, where $(x_{1,n}, x_{2,n})$ are the closest points after the n^{th} iteration of the algorithm. For each pair of points the Kuhn-Tucker conditions are satisfied on one object because the vector between the points is normal to the surface of one of the objects. Zeghloul and Rambeaud solve the 'zig-zagging problem' as follows: if the Kuhn-Tucker conditions are only satisfied for object O_1, then instead of setting the search direction for O_2 to null, it is set to the

[2]as described by Zeghloul and Rambeaud in their earlier 1992 paper [81].

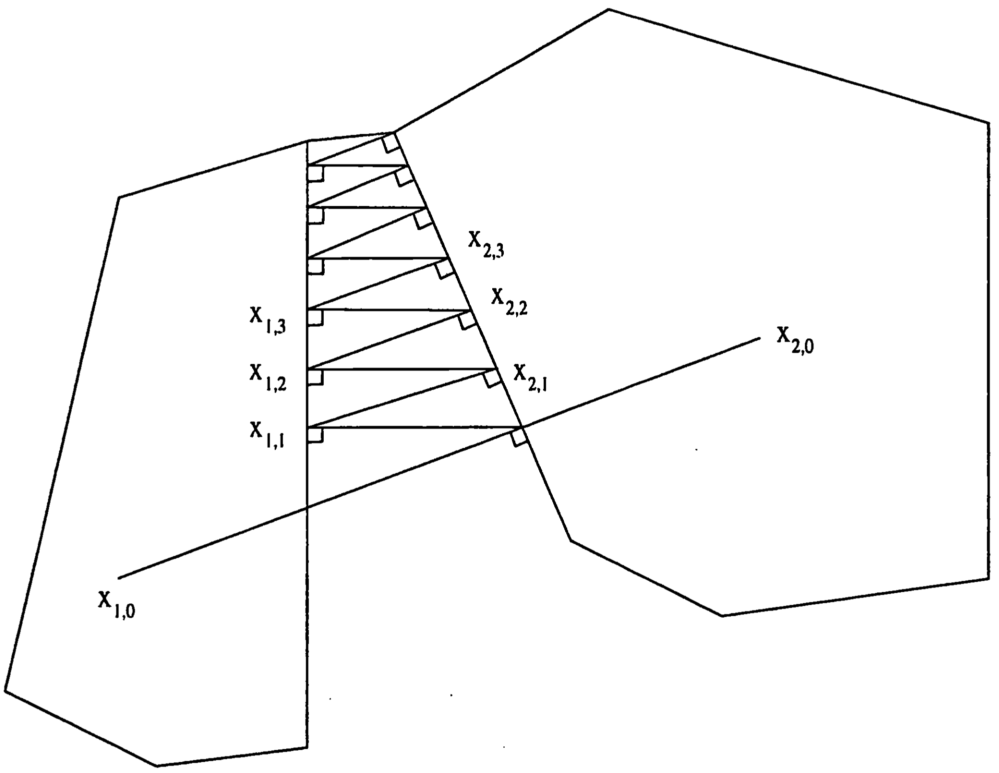


Figure 5.8: The zig-zagging problem of Bobrow's algorithm.

projection of the search direction of the other object onto the planes that x_1 lies on.

Zeghloul and Rambeaud also suggest a change to the test for whether the Kuhn-Tucker conditions are satisfied. Geometrically, the Kuhn-Tucker conditions are satisfied for x_1 if x_2 lies in the cone of vertex x_1 and spanned by the normals of the planes that x_1 lies upon. As Zeghloul and Rambeaud state, this can be tested with n dot-product evaluations, where n is the number of planes that x_1 lies on. Figure 5.9 gives a two-dimensional example of the cone.

Turnbull and Cameron (1998) [74] give a minimum distance algorithm for NURBS-defined convex objects. It is based on Gilbert, Johnson and Keerthi [29] and again uses support properties. A method of calculating the support properties for NURBS objects is given. The algorithm has been implemented for two-dimensional shapes and formulae have been developed for three dimensions.

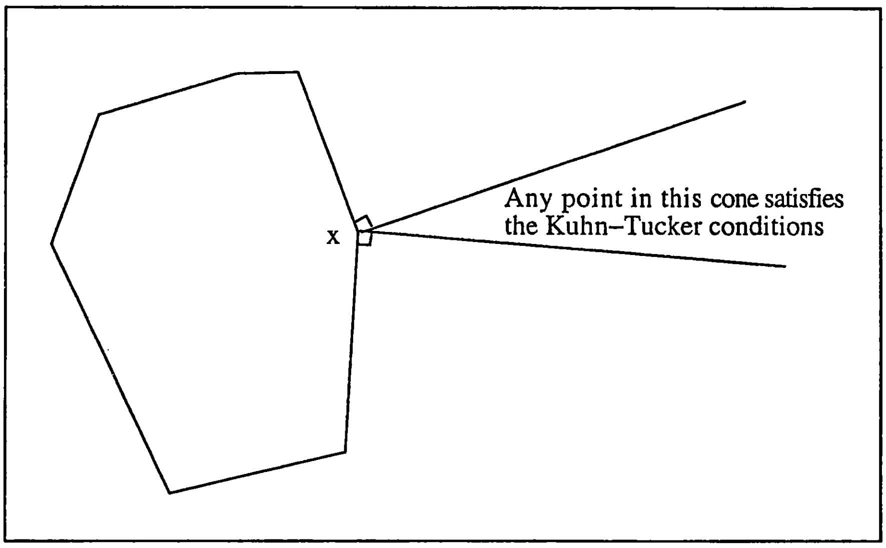


Figure 5.9: The cone for which the Kuhn-Tucker conditions are satisfied.

5.2.4 Non-convex polyhedra

With the exception of Chin and Wang's algorithm for planar polygons, the minimum distance algorithms described in the foregoing section use minimizing techniques to find the closest distance which rely on the convex property of the objects. There is little research on minimum distance between non-convex objects as this is a much harder problem.

Red (1983) [61] gives an algorithm for finding the minimum distance between two polyhedra. However, concave polyhedra must be stored as unions of convex polyhedra, with no intersection allowed between the convex polyhedra, so essentially Red's algorithm is a technique for finding and merging the minimum distances between convex polyhedra. Red anticipates the application of the algorithm as robot task simulation, and the paper considers the problem of finding the distance between polyhedra representing a manipulator and polyhedra representing obstacles. Red suggests two global elimination strategies to reduce the number of polyhedron to polyhedron comparisons; he also describes a local strategy for eliminating unnecessary polygon-to-polygon comparisons, and an algorithm for finding the minimum distance between two polygons in three-dimensional space.

Red's first global elimination strategy consists of using spheres or rounded cylinders (spherical extensions of lines) to bound all polyhedra. Only the bounding shapes closer together than a 'sensitivity distance' are considered for distance checking. Red suggests that this distance can be set manually for applications where a user is planning a manipulator path, and that a value based on the proximity of the closest object should be used when the manipulator is far from obstacles in a configuration mapping application. Red's other global strategy is that of ignoring obstacles outside the volume that could be swept by the manipulator.

The minimum distance between polyhedra is found by computing the distances between their constituent polygons and taking the minimum value. Red's basis for reducing the number of comparisons is that if both polyhedra are convex, then non-facing polygons do not need to be compared, thereby eliminating unnecessary polygon-to-polygon comparisons. Red's method for finding the minimum distance between two polygons in three-dimensional space considers the relationships between the polygons F_i and F_j and the planes P_i and P_j in which the polygons lie. These relationships are classified and the minimum distance between the polygons is found by projecting F_i onto P_j and F_j onto P_i.

Red asserts that computational times increase linearly or less than linearly with the number of polyhedra.

5.3 New Research Completed

To find the minimum distance between two set-theoretically defined objects, I have used a spatial division approach that recursively divides the object as described in section 2.2.2, and finds the two closest sub-model's boxes: the minimum distance between these boxes is returned as the minimum distance between the objects. An initial naive approach to this was developed to give a basis for the comparison of accuracy and speed of more sophisticated methods that might subsequently be developed. This 'brute force' method (described in section 5.3.1) recursively divides a model to a certain smallest size and then finds the two closest boxes. A more refined approach was later developed. This is laid out in section 5.3.3. In it a model is recursively spatially divided and the minimum distance between sub-model's boxes is found while model division is performed. The model is divided more finely in the areas of the model where the closest points are considered likely to occur, depending upon information found so far.

5.3.1 The brute force approach

This simple approach is useful for checking the accuracy of results obtained by more advanced methods. The model is first divided using a simple division procedure that divides sub-model boxes in half along their longest side if they contain some of the surface of the object and are not 'too small'. The definition of 'too small' for a sub-model is that the ratio of its volume to the volume of the box that contains all of the objects (the model's box) is smaller than a given value (usually of the order of 10^{-6}).

The two leaf boxes containing surface of different objects with closest centroids

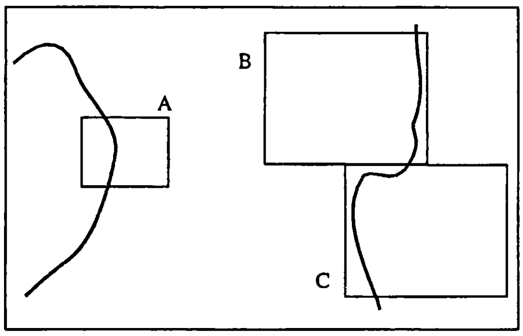


Figure 5.10: The distance between boxes.

are found in the divided model by finding the distance between the centroid of each leaf box containing surface of the first object and the centroid of each leaf box containing surface of the second object. The minimum distance between the two objects is set to the distance between the centroids of these two closest leaf boxes. However, this method does not always give accurate results, as can be seen in the example in figure 5.10 where the distance between the centroids of boxes A and B is smaller than the distance between the centroids of A and C, although the objects (shown by the curves) are actually closer in C and A than B and A. To ensure more accurate results, the model has to be divided into smaller boxes.

Finding the two closest boxes needs mn comparisons, where there are m leaf boxes containing object 1 and n leaf boxes containing object 2; m and n are dependent upon the smallest box size and the complexity of the surfaces of the objects. This method therefore has computational efficiency of $O(mn)$.

5.3.2 Results

The accuracy of this method depends on the size of the smallest boxes. If the boxes are assumed to be cuboids, then the largest possible error is $2\sqrt{3}\sqrt[3]{fV}$

Model	Minimum size ratio	Distance found	Actual distance	Actual error /%	Largest possible error /%
Two cuboids	10^{-3}	0.89	1.50	41	230
Two cuboids	10^{-4}	1.30	1.50	13	107
Two cuboids	5×10^{-5}	1.48	1.50	1.3	85
Two cuboids	10^{-5}	1.42	1.50	5.3	50
Mirrored cuboids	10^{-3}	5.03	5.20	3.3	67
Mirrored cuboids	10^{-4}	5.01	5.20	3.3	31
Mirrored cuboids	5×10^{-5}	4.82	5.20	7.3	25
Mirrored cuboids	10^{-5}	5.01	5.20	3.7	14

Table 5.1: Accuracy of results for brute force distance procedure, with distances given in model units.

where f is the minimum size ratio, i.e. the ratio of the volume of the smallest box to the volume of the largest box, and V is the volume of the model's initial all-encompassing box. The error is proportional to $\sqrt[3]{f}$, which means that the smallest box volume must be greatly reduced to give even a modest increase in accuracy, as would be expected for a volume-distance relationship. Table 5.1 shows the distance found and the maximum possible error for models 'Two cuboids' and 'Mirrored cuboids' (shown in figure 5.11) for different minimum size ratios. Note that in all cases the actual error is significantly smaller than the maximum possible error. Two anomalies where the error increases for smaller box size are evident for 'Two cuboids' at minimum size ratio 10^{-5} and 'Mirrored cuboids' at minimum size ratio 5×10^{-5}. This is because the distance calculated is the distance between the centroids of the boxes, and the larger boxes were centred closer to the actual closest points of the two objects than were the smaller boxes. This is illustrated by the two-dimensional example of figure 5.12, where the distance between the centroids (shown by the arrowed line) of the two larger boxes is closer to the actual distance between the objects than the distance between the centroids of the smaller boxes.

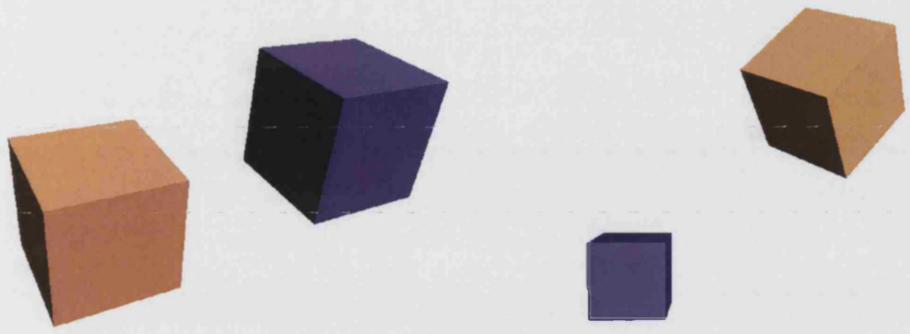


Figure 5.11: Sample models for the brute force minimum-distance method.

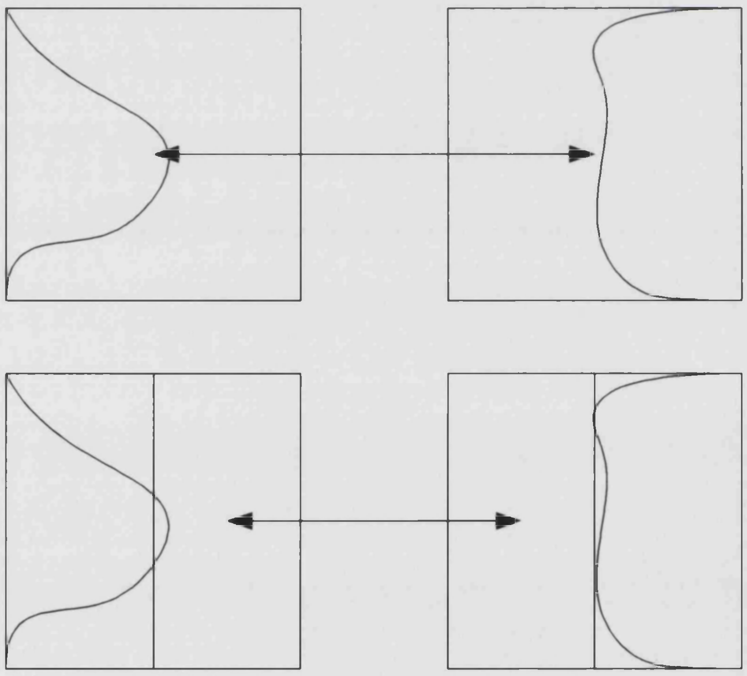


Figure 5.12: The distance between the centroids of large boxes can be more accurate than that between the centroids of smaller boxes.

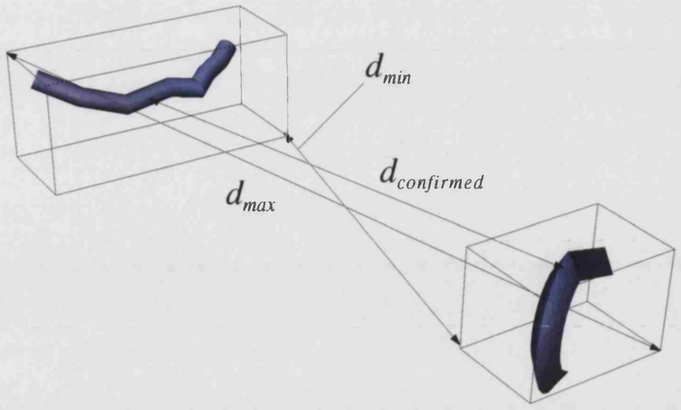


Figure 5.13: The terms d_{min}, d_{max} and $d_{confirmed}$ illustrated.

5.3.3 Minimum distance using adaptive division and distance bounds tracking

I have researched a more refined approach [57], whereby the model is divided more intelligently, concentrating on boxes that are likely to contain the closest points, and ignoring other boxes. As the algorithm progresses the model is divided more finely and the calculation of the distance improves.

For the purposes of explaining the algorithm, five terms are used: d_{min}, d_{max}, $d_{confirmed}$, D_{lower} and D_{upper}. These are defined below, followed by further explanation of the steps of the algorithm. Figures 5.13 and 5.14 show the geometric meaning of the terms for example three- and two-dimensional models.

d_{min} A distance less than, or equal to, the minimum distance between two sub-models. This is usually the minimum distance between the boxes, but the exact smallest distance between the sets is used if it can be calculated. The current implementation finds the exact distance only for convex polyhedral sets, each made up of three planes, using a minimization technique.

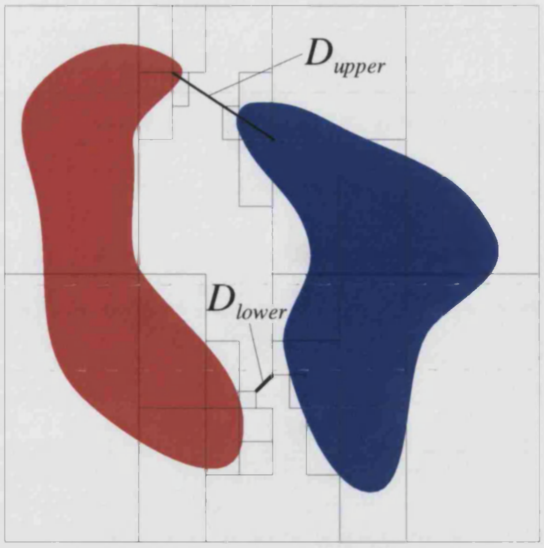


Figure 5.14: The terms D_{lower} and D_{upper} illustrated.

d_{max} The maximum distance between two boxes.

$d_{confirmed}$ The distance between two points in solid parts of the sets of two sub-models. This is not necessarily the minimum distance between the sub-models, but it is guaranteed to be equal to or larger than the minimum distance. The process for calculating $d_{confirmed}$ is described below.

D_{lower} The smallest value of d_{min} for any pair in the group of candidate pairs. It is the lower bound on the minimum distance, and its initial value is zero.

D_{upper} The smallest value of $d_{confirmed}$ for any pair in the group of candidate pairs. It is the upper bound on the minimum distance and is initially set to the diagonal of the original model's box, so that it is not less than the minimum distance between the two objects.

Calculating $d_{confirmed}$ for a pair of sub-models

Smaller values of $d_{confirmed}$ reduce the value of D_{upper}, which in turn reduces the minimum distance bounds; this causes more sub-model pairs for which $d_{min} >$

D_{upper} to be discarded (see below).

If a pair of sub-models contain simple sets between which the minimum distance can be exactly calculated, $d_{confirmed}$ is set to the value of the minimum distance. For more complicated sets, a point on the surface or in the solid part of each model is sought, and $d_{confirmed}$ is set to the distance between the points. Closer points result in more accurate values of $d_{confirmed}$. If such points cannot be found, then $d_{confirmed}$ cannot be calculated for the sub-models.

In my implementation of the algorithm, the distance between convex sets with three planes is calculated by using a simpler version of Bobrow's minimization technique [6]. This could be extended to more complicated sets if the distance between the sets could be exactly calculated: cylinders and spheres would be suitable sets to study first.

To find points on the surface or in the solid part of each model, a Newton-Raphson root-finding method is used. If no such points are found then the centroid and the corners of the sub-model's box are tested. The Newton-Raphson root-finding method is used by finding a root of $\sum_{i=1}^{n} r_i^2$ where the r_i are the primitives' potential functions and n is two for an edge and three for a corner. The r_i^2 are normalized so that high-degree primitives do not have more influence than lower-degree primitives. The Newton-Raphson method needs a point, p_0, from which to start converging[3]. The point p_0 is set to the centroid of the model's box, or if that is not in solid, to one of the corners of the model's box that is in solid. If two points that are in solid parts of the set cannot be found, then $d_{confirmed}$ cannot be found for the current pair of sub-models so this step of the algorithm is skipped and D_{upper} will not be reduced.

[3]If the point p_0 is in the solid part of the sub-model, then the point found by the Newton-Raphson method is also more likely to be in the solid part of the sub-model.

Outline of the algorithm

Given a model, M_0, containing two labelled sets, S_1 and S_2, the algorithm finds lower and upper bounds for the minimum distance between S_1 and S_2. The algorithm can be expressed as five stages, summarised here and explained in more detail later.

1. Using recursive spatial division, divide M_0 into several sub-models, each sub-model containing only one of the sets S_1 or S_2. If any sub-model has been divided to a certain smallest size and it still contains both S_1 and S_2, terminate the algorithm with 'sets are touching'.
2. Categorise the sub-models as containing S_1 or S_2.
3. Create a group[4]of candidate pairs, each pair consisting of one sub-model from each category.
4. Remove one pair of sub-models from the group of candidate pairs and divide one of the sub-models in this pair. For each of the two new pairs created by this division, if it is possible that the pair could be the closest pair, then add this pair to the group of candidate pairs and also update the value of D_{upper}.
5. If any of certain terminating conditions (see below) have been met, terminate the algorithm, otherwise return to Step 4.

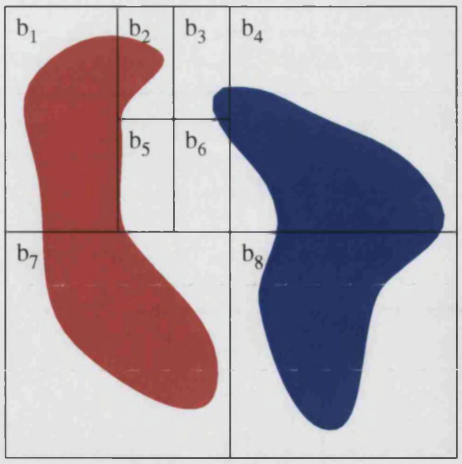


Figure 5.15: A model after division to find boxes that only contain part of one of the sets S_1 and S_2.

Step 1: divide M_0 into sub-models

The model is divided recursively into sub-models which each contain part of only one of S_1 or S_2. Figure 5.15 shows a two-dimensional example of such a division. If any sub-model has been divided so many times that its volume is smaller than a threshold volume[5], and it still contains parts of both S_1 and S_2, then the sets are considered to be so close that they touch or interpenetrate, and the algorithm is terminated and returns this result.

Step 2: categorise the sub-models

The label of the set in each sub-model is retrieved and compared with the known labels for S_1 and S_2. If the label of the set is equal to the label for S_1 then add the sub-model to the collection of sub-models K_1; otherwise if the label is equal to the label for S_2 then add the sub-model to the collection of sub-models K_2.

[4]A set of candidate pairs would be a more accurate term, but to avoid confusion with sVLIs sets, the word group is used in this thesis.

[5]This is typically set to 10^{-6} times the volume of the original model.

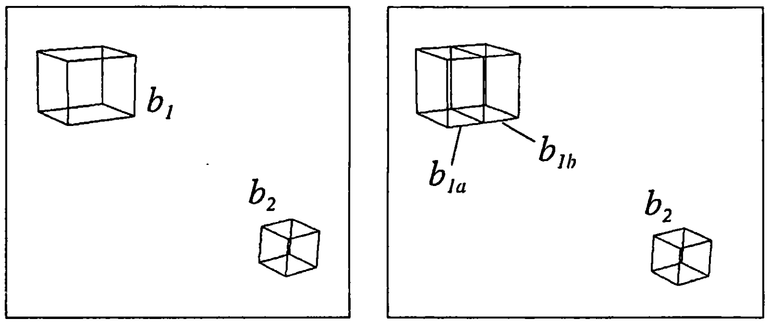


Figure 5.16: Dividing b_1 - one of a pair of sub-models' boxes (b_1, b_2) to create the boxes b_{1a} and b_{1b} and corresponding sub-models.

Step 3: a group of candidate pairs is created

For each possible pair of sub-models where one sub-model is in K_1 and the other is in K_2, check the values for that pair by using the function described below in the section 'Function to check values for a pair of sub-models', on page 116.

Step 4: remove one of the sub-model pairs and further divide one sub-model of the pair

A pair of sub-models $\{M_1, M_2\}$ is removed from the group of pairs. One of them is spatially divided into two smaller sub-models, M_{ia} and M_{ib}, in the manner described earlier and as illustrated in figure 5.16. Two new sub-model pairs $\{M_{ia}, M_j\}$ and $\{M_{ib}, M_j\}$ are created. The 'Function to check values for a pair of sub-models', described below on page 116, is then applied to the two new pairs to decide whether they are added to the group of candidate pairs and also to update the value of D_{upper}.

The order in which the sub-models are divided has a great effect on the efficiency of the algorithm. To minimize execution time, sub-model division strategies are applied which increase D_{lower} and decrease D_{upper} to narrow the bounds on the minimum distance, as explained in the following paragraphs.

The first pair of sub-models to be removed from the group of pairs is the pair with the smallest value of d_{min}. Of the two sub-models within this pair, the sub-model with the largest box is divided. On the next iteration of the algorithm, the pair marked as the 'upper-bound pair' is removed and the sub-model with the largest box within that pair is divided. This order of alternating between removing and dividing the pair with the smallest value of d_{min}, and the pair marked as the upper-bound pair, is repeated until the algorithm terminates.

When a box b_1 in the pair (b_1, b_2) that is marked as the 'upper-bound pair' is divided into two sub-boxes b_{1a} and b_{1b}, if the sub-model with the box b_{1b} contains part of a set, and if the value of $d_{confirmed}$ for the closest of the two new pairs of boxes $\{b_{1b}, b_2\}$ is smaller than $d_{confirmed}$ for the pair $\{b_1, b_2\}$ that was divided, then D_{upper} will have been reduced. If the pair $\{b_{1b}, b_2\}$ is again marked as the upper-bound pair it will be further divided in later iterations of the algorithm, and D_{upper} will be quickly reduced with subsequent iterations. Reductions in D_{upper} will reduce the minimum distance range, and will also lead to those pairs for which $d_{min} > D_{upper}$ being eliminated from the pair structure in future.

When a box in the pair with the smallest d_{min} is divided, if $d_{min} > D_{lower}$ for the two newly-created pairs, then D_{lower} may have changed. Under these conditions, the smallest value of d_{min} for all of the pairs in the candidate pair group is found, and D_{lower} is then given that value.

Dividing one box of the pair with the smallest d_{min} will increase the lower bound on the distance only if the closest sub-model created (b_{1b}) does not contain part of a set. The new smallest d_{min} will either be the d_{min} for $\{b_{1a}, b_2\}$, or the d_{min} for another pair, if that is smaller. For many models, the sub-model's pairs found in the initial 'single set' division will have boxes that touch each other (for example, $\{b_2, b_3\}$ in Figure 5.15) and therefore have a d_{min} of zero. In this case the lower bound on the distance will also be zero. All of these 'touching' sub-model pairs must in reality have air between the sets in those sub-models (otherwise they would have been divided to the minimum volume in the 'single-set' division and the algorithm would have terminated with the result that the sets S_1 and S_2 were touching). After repeated division of their closest boxes, those boxes will be found to contain air, and will be removed from the pair structure. Some of the pairs that have small values of d_{min} may contain sets that are actually widely separated, and after division the new value of d_{min} for the sets could be larger than D_{upper}, and the pair would not be added to the pair structure.

If there are no more touching sub-model pairs and the inner pair created by dividing the pair with smallest d_{min} contains surface, that pair will be divided again. This will be repeated until the inner pair no longer contains surface, at which point only the outer pair will be added to the candidate pair list. This outer pair will have a d_{min} larger than the d_{min} of the original pair, and so will not necessarily be the pair divided in the next division.

Step 5: termination of the algorithm

When the algorithm terminates, the bounds on the minimum distance have been found and are given by D_{lower} and D_{upper}.

The terminating conditions are designed to allow the algorithm to continue until a final value for minimum distance has been found within specified tolerances, as well as to make sure the algorithm does not continue indefinitely, with only a marginal improvement in the minimum distance range computed. Terminating conditions can be combined so that the algorithm terminates only when two or more conditions are true.

Possible conditions that ensure that the distance found is a reasonable measure of the true minimum distance are:

- The range of the minimum-distance bounds $(D_{upper} - D_{lower})$ is smaller than a given size.
- The ratio of the range $D_{upper} - D_{lower}$ to the midpoint of that range is smaller than a given value.
- The largest sub-model's box still in consideration is smaller than a given size.
- There are fewer than a given number of candidate pairs.
- There are no touching sub-models $(D_{lower} > 0)$.

Possible conditions that ensure the algorithm will not continue indefinitely are:

- There has been a negligible reduction in the size of the minimum-distance bounds for a given number of divisions.
- A given number of divisions have been performed.

Function to check values for a pair of sub-models (Used in steps 3 and 4)

This function decides whether a pair should be stored in the group of candidate pairs; if the pair should be stored, it updates the value of D_{upper}. It uses the following steps:

- Firstly the values of d_{min} and d_{max} are found for the pair of sub-models. To calculate d_{min}, if the smallest distance between the sub-models can be calculated exactly using my implementation of Bobrow's minimization approach [6], then use that for d_{min}. Otherwise $d_{min} is$ set to the minimum distance between the sub-models' boxes. d_{max} is calculated as the distance between the furthest corners of the sub-model's boxes. Some investigation was done on a more exact calculation of d_{max}, but it was found that this did not improve the overall efficiency of the algorithm, as shown later in section 5.3.4.

- If $d_{min} > D_{upper}$ the pair cannot contain the closest points of S_1 and S_2, because D_{upper} is larger than or equal to the minimum distance between S_1 and S_2. If this condition is true, the pair is not added to the group of candidate pairs and this function is exited.

- Add the pair to the group of candidate pairs.

- If $d_{max} < D_{upper}$ for the pair of sub-models the pair may contain the closest points between S_1 and S_2 so $d_{confirmed}$ is calculated for the pair. If $d_{confirmed} < D_{upper}$, then D_{upper} is set to $d_{confirmed}$ and the pair is marked as the 'upper-bound pair'.

A pair for which $d_{min} < D_{upper} <= d_{max}$ may contain the closest points, and a value of $d_{confirmed}$ could be found for the pair that is less than D_{upper}. In practice, calculating $d_{confirmed}$ for all these pairs takes more time than letting the algorithm run for more iterations to gain a result of the same accuracy.

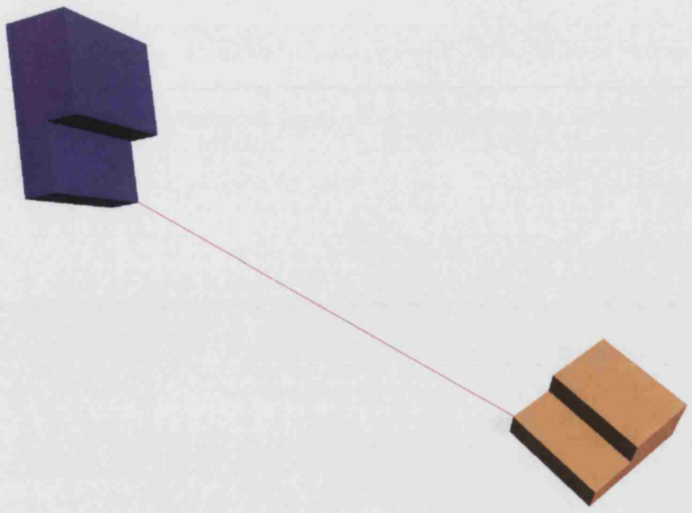


Figure 5.17: Minimum distance for 'L shapes' model.

The way in which the group of candidate pairs is stored affects the speed of the algorithm. I have chosen to implement the algorithm using a binary search tree of sub-model pairs, because a binary search tree has fast insertion, removal and searching, which are essential operations for the algorithm.

5.3.4 Results

All times in this section are CPU time measured on a Silicon Graphics Onyx with two 150MHz R4400 processors.

To demonstrate visually the results of the algorithm, the minimum distance found for some of the sample models is shown in figures 5.17 to 5.20, with the separate objects shown in different colours. A red line connects the points between which the minimum distance lower bound (D_{lower}) has been found. Additionally on the 'Sine curves' model, a blue dashed line indicates the points for the minimum distance upper bound (D_{upper}).

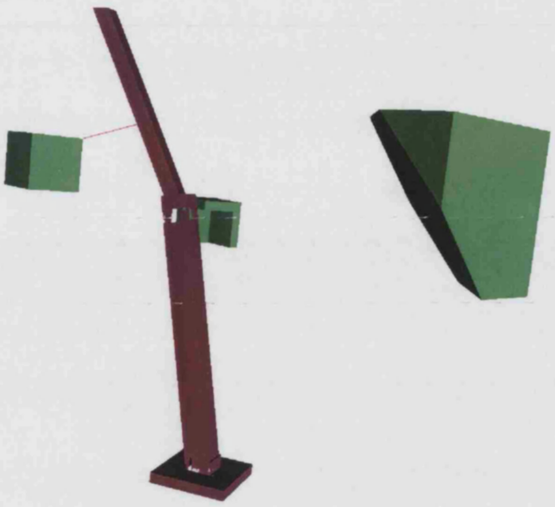


Figure 5.18: Minimum distance for 'Robot' model.

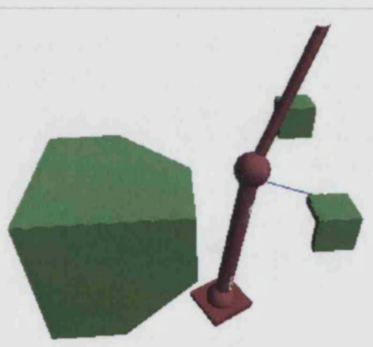


Figure 5.19: Minimum distance for 'Non-polyhedral robot' Model.

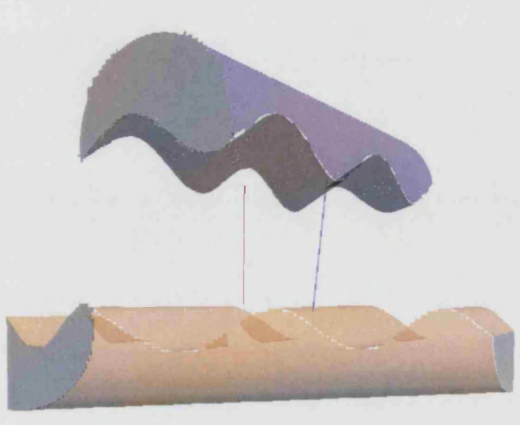


Figure 5.20: Minimum distance for 'Sine curves' model.

Model	Actual distance	Brute force distance	Minimum-distance bounds
Two cuboids	1.5	1.42	1.5000-1.5000
Mirrored cuboids	$\sqrt{27} \simeq 5.1962$	5.01	5.1678-5.1964
Racing car	0.8062	0.71	0.7906-0.8067
Robot	1.0542	1.09	1.0304-1.0602
Non-polyhedral robot	-	1.01	1.0886-1.2062
Sine curves	-	5.94	5.9838-6.0148
Lever	-	7.74	7.8170-7.9931
'L' shapes	$\frac{14}{\sqrt{3}} \simeq 8.0829$	8.04	8.0629-8.0890

Table 5.2: Accuracy of the minimum-distance algorithm

Accuracy of the minimum-distance algorithm

To verify that the results obtained by the minimum-distance algorithm are correct, the results were compared with either the exact distance if it could be calculated (by examining the models), or the distance calculated by using the brute-force method with divided box volume of 10^{-6} times the volume of the model's box. For each model, table 5.2 shows one or both of these distances, and the minimum-distance bounds computed by the 'adaptive division and distance bounds tracking' algorithm. The condition used to terminate the algorithm in these tests was that the ratio of the volume of the smallest sub-model's box to the volume of the original model's box was less than 10^{-9}.

The bounds given by the minimum-distance algorithm are accurate for the sets with known distances. For those, the actual distance is close to the upper bound (D_{upper}) because D_{upper} is calculated exactly. For the other sets, the bounds are close to that given by the brute-force method. The bounds do not contain the distance calculated by the brute force method, but that is not guaranteed to give the correct result as can be seen by comparing the actual distance with the distance calculated by the brute force method for those models for which the actual distance is known, so this does not imply that the 'adaptive division and distance bounds tracking' algorithm is inaccurate.

Investigation into how properties of the models affect the time taken to calculate the minimum distance

Several sets of tests were performed to investigate how the time taken to calculate the minimum distance is affected by the number of primitives in the model, the algebraic complexity of the primitives, the 'crinkliness' of the models, and the distance between the objects. For each of these properties, the minimum distance between the models of table 5.2 was found with the two objects within the models rotated by a random angle with the centre of rotation located within the object. At least seven minimum-distance calculations were performed for each model; for each calculation the model was rotated differently. The division resolution was 10^{-9} for all tests.

a) Number of primitives

Figure 5.21 shows how the number of primitives in the sets affects the time taken to find the distance between two sets. No trend can be seen in the data, therefore the number of primitives does not have any significant effect on the time taken to find the minimum distance, which is very encouraging.

The values for two data points with times of greater than 250s have not been plotted to make the graph more readable. These points were for the sine curves model which has four primitives and took 250s and 1060s to find the minimum distance.

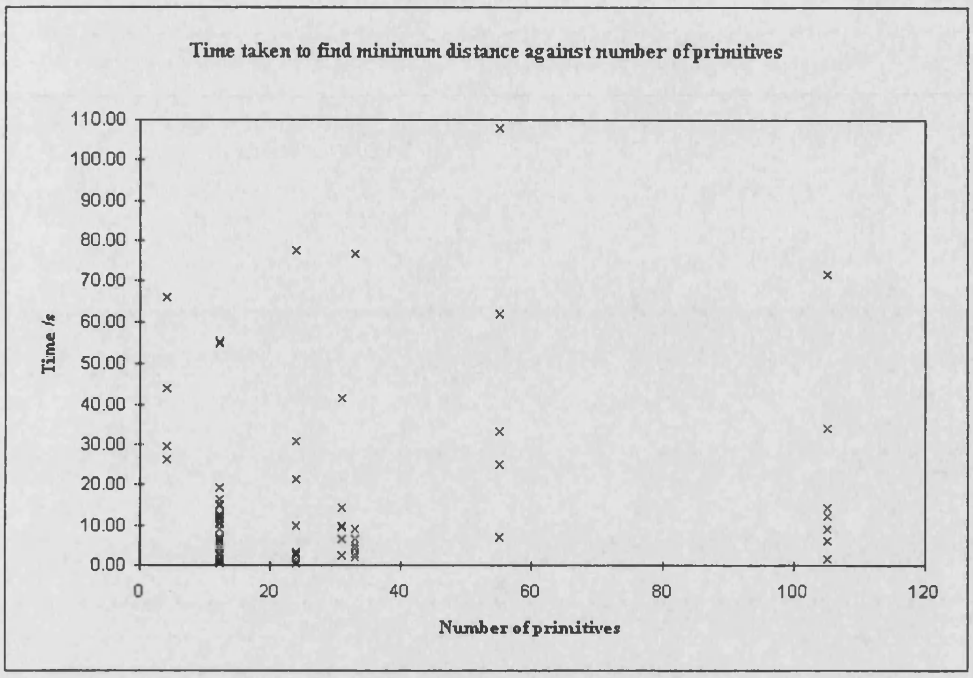


Figure 5.21: Time taken to find the minimum distance between objects with different numbers of primitives

b) Algebraic complexity of the primitives

Figure 5.22 shows how the time taken to find the distance is affected by the order of the highest degree primitive in a model. The Trig values are those for the 'Sine curves' model and have been placed at the right hand side of the graph for convenience and not to denote that they are of degree greater than four.

The time taken to calculate the minimum-distance increases approximately linearly with the order of the highest degree primitive, with the exception of the degree 4 model.

c) Crinkliness

Crinkliness [54] for a three-dimensional object is defined as the ratio of the surface area of the object to the surface area of a sphere with the same volume as the

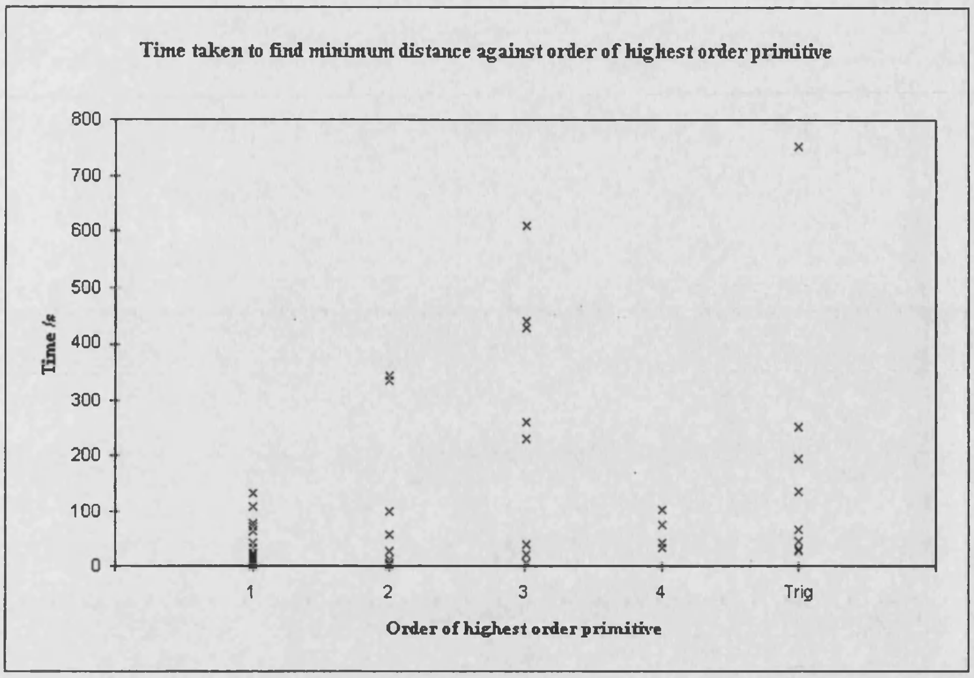


Figure 5.22: Time taken for models with different algebraic complexities

object; crinkliness can be used to measure how 'irregular' the surface of an object is. A ball will have a crinkliness value of one, and a starfish will have a relatively high crinkliness value. Figure 5.23 shows the time taken for the example sets plotted against the sum of the crinkliness of the sets.

The two tests for which the minimum-distance was computed in more than 250s have again not been plotted. They give points of (2.7, 250) and (2.7, 1060). There is no general trend in the data, so it is assumed that crinkliness has little effect on the time taken to find the minimum distance.

d) Distance between the objects

It was thought that the separation between the objects may affect the time taken to find that distance. Figure 5.24 is a plot for two sets of the translation from the start position against the time taken to find the distance. More positive

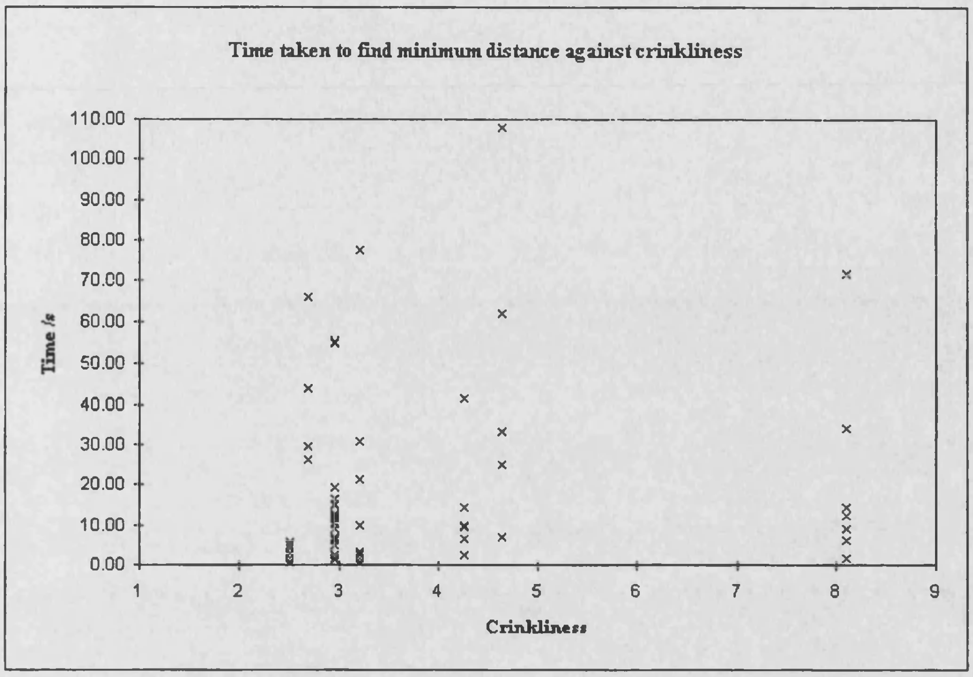


Figure 5.23: Time taken to find minimum distance for models with different crinkliness values.

translations mean that the objects are further apart. The results for the 'sine curves' models are shown as o, the results for the cuboids mirrored diagonally are shown as +. There is no general trend in the data.

The results show that three of these four properties of the models have little effect on the time taken to execute the algorithm, suggesting that such measurement is unable to account for a binary tree structure that is divided dynamically depending upon the geometrical structure of the model rather than the structure of the data, as can be used, for example, to classify computational complexity of point-set algorithms by the number of input points.

Methods of calculating d_{max}

The value of d_{max} is calculated for a pair of sub-models in the 'function to check values for a pair of sub-models' by finding the distance between the sub-model's

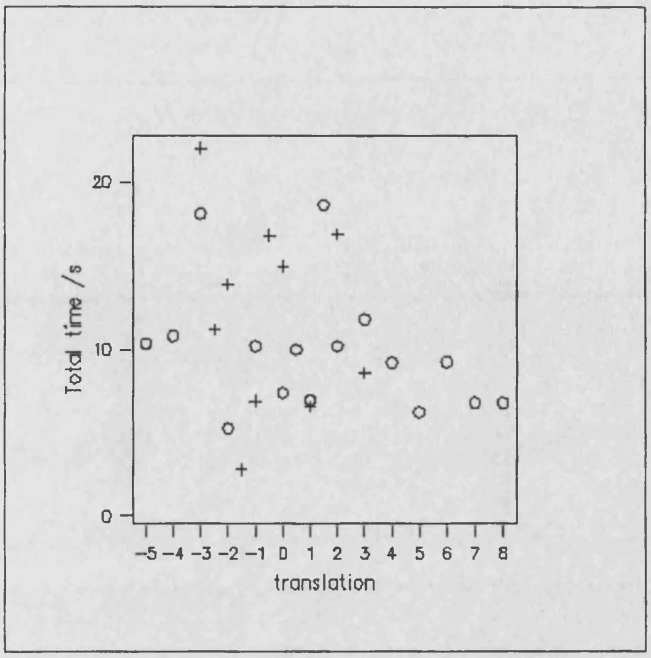


Figure 5.24: Time taken to find the minimum distance for objects separated by different amounts

boxes furthest corners. I have investigated the efficiency implications of using other methods of calculating this, namely:

- The distance between the centroids of the sub-models' boxes if the centroids are in solid.
- The distance between the closest corners of the sub-models' boxes that membership test as solid.
- Using the Newton-Raphson root-finding method provided by sVLIs to find a point in each sub-model and calculate the distance between the two points found. This only works on sub-models with two or three primitives and has been explained earlier in the section 'Calculating $d_{confirmed}$ for a pair of sub-models'.

Method used to calculate d_{max}	Racing car	Robot
Furthest corners (reference value)	20.9	15.0
Centroids	24.4	15.5
Closest solid corners	25.1	17.4
Newton-Raphson	21.3	15.8
Centroids and closest solid corners	24.6	17.6
Closest solid corners and Newton-Raphson	27.1	21.8
Centroids, closest solid corners and Newton-Raphson	30.1	18.6

Table 5.3: Time in seconds to find the minimum distance using different methods to decrease d_{max}

The minimum distance algorithm was applied to two sample models: 'Racing car' and 'Robot with obstacles' (shown in figures 4.15 and 4.16 from page 71). The time taken to find the distance is shown in table 5.3 for each method and combinations of the methods.

All of the more exact calculation methods make the minimum-distance algorithm slower than the 'furthest corners' approach, with the Newton-Raphson method having the smallest time overhead. This shows that the reduced number of sub-model pairs that need to be considered in the 'function to check values for a pair of sub-models' to calculate $d_{confirmed}$ does not offset the additional overhead of more complicated methods of calculating d_{max}.

Summary of results

- The proposed 'progressive division' minimum-distance algorithm has been shown to be accurate by comparing its results to those obtained by exact calculation where that has been possible, or by comparison with results from the brute force minimum-distance algorithm.
- Neither the total number of primitives in the models, the crinkliness of the objects, or the distance between the objects has any noticeable effect on

the time taken to find the minimum distance.

- The time taken to find the minimum distance increases approximately linearly with the order of the highest degree primitive in the models.
- Methods of calculating d_{max} for a pair of sub-models that are more sophisticated than calculating the distance between the furthest corners of the sub-model's box do not improve the overall efficiency of the algorithm due to the large overhead of performing the more complicated calculations.

5.4 Future work

5.4.1 Improvements to the existing minimum-distance algorithm

The main way of improving efficiency of the algorithm would be to reduce the number of candidate pairs to be considered. Possible approaches include checking the d_{min} value of pairs before division, finding a smaller initial value for D_{upper} and an exact distance calculation for simple pairs of sub-models.

Before dividing a sub-model M, it may be worth checking whether the d_{min} values for all of the pairs $\{M, M_i\}$ (where i is the value for each of the partners of M) are greater than D_{upper} and, if they are, then that sub-model can be discarded. It is possible that the overhead of performing this check may outweigh the benefit of removing the sub-model and its partners from the candidate sub-model pairs.

Finding a smaller initial value for D_{upper} would mean that fewer sub-model pairs may be added to the group of candidate pairs because their $d_{min} > D_{upper}$. It would be possible to find the distance between the two sets by using a minimization technique such as Bobrow's [6]. This technique requires that the sets are convex polyhedra to find the minimum distance between them. However, even if they are not, it will still find two points on the surfaces of the sets which would be closer than the diagonal of the model's box (the current initial value of D_{upper}).

The sVLIs function that evaluates the contents of a model uses interval arithmetic and is conservative, i.e. it may report that a model contains some boundary when it is actually all air or all solid. If an exact test for whether a model was air, solid or surface existed, then these sub-models that the current evaluation regards as containing surface, but are actually air, could be eliminated from further consideration. For polygonal sub-models it is possible to compute whether there is really any surface in a sub-model; Voiculescu [75] is researching techniques to evaluate more accurately whether a model more complicated than a simple polyhedron contains surface by using the model's algebraic properties. Milne [53] has also done some work on this problem in his Geometric Algebra System.

Exact distance calculation for simple pairs of sub-models

Finding the exact minimum distance between a pair of simple sub-models could make the algorithm more efficient. It would set the value of $d_{confirmed}$ to the lowest possible value for that pair, which may decrease D_{upper}, in turn meaning that less sub-model pairs need be considered in future.

For a pair of sub-models that consist only of convex polyhedral objects, Bobrow's algorithm [6] could be used to find the exact distance. The sub-models would generally be open sets, which Bobrow's algorithm is not designed to work with, but the planes of the sub-model's box could be used to bound the sets. Implementation issues would involve:

- finding start points,
- testing the Kuhn-Tucker conditions,
- finding new search directions,
- finding the amount to move in the search direction.

Each start point could be any solid point in the object, which could be simply found by dividing the model until a solid box was located: any point within this box could be used. Finding the planes that the points lie on, in order to test the Kuhn-Tucker conditions, could be done with point-membership tests on all the planes in a small box around the point. The new search direction could be ascertained by projecting the vector between the points onto a plane or planes. It may be possible to find the distance to move by tracing a ray until it leaves the surface of the object.

Calculating the exact distance between non-polyhedral sub-models would make the method more efficient for non-polyhedral objects. Spheres and cylinders, as spherical extensions of points and lines, should prove to be relatively straightforward objects to begin work upon.

If models are constructed only from convex primitives, an efficient way to check if a sub-model were convex would be to check if the set is stored in *disjunctive normal form (DNF)*, i.e. the intersection operator never appears above any union operator in the set-theoretic tree. In general, convex parts of models will be stored in DNF as that is usually the result of constructing them in a straightforward manner. It is possible to convert any set-theoretic tree into a union of DNF [35] trees, but in the worst case, the size of the tree grows exponentially with increasing numbers of leaf nodes of the original tree. Specifically, if the original tree has $2K$ leaf nodes and it has intersection operators everywhere but at the lowest internal nodes, the DNF will have 2^K product terms, each of which has K leaf nodes.

5.4.2 Breadth-first division for minimum-distance calculation

The basis of this method for computing the minimum distance between two objects was proposed to me by Wise [76]. Assume that a function exists that can quickly calculate the distance between an axially aligned cuboid (box) and an object in a set theoretically defined model. Use adaptive spatial division in a breadth-first[6] manner to create a sub-model tree, and at each division of a sub-model, calculate the intervals for distances between the sub-model's box and each of the objects in the model. Let $[d_{lo}, d_{hi}]$ be the sum of the squares of these two intervals. If the d_{lo} is less than the 'best' value found, then set the 'best' value to d_{lo}. Otherwise, if d_{hi} is greater than the 'best' value found, then mark the sub-model so that it is not divided further in the spatial division.

This method searches a divided model while the model is being divided to find the smallest value of d_{lo}. The search avoids local minima as it is a global search, and is made efficient by discarding areas of the model which have values of d_{hi} larger than the smallest value of d_{lo}.

The search would continue indefinitely so terminating conditions would need to be used. If an accurate result were required, terminate the search when the

[6]i.e. divide all of the models at one level of the sub-model tree once before dividing any model at the next level of the tree.

interval $[d_{lo}, d_{hi}]$ for the sub-model with the smallest value of d_{lo} is less than a certain threshold value. For a faster search, terminate when the number of levels in the divided sub-model tree exceeds a threshold value. In practice a balance of the two conditions will probably have to be used. When the search terminates the minimum distance between the objects is given by $\sqrt{d_{lo}}$.

5.5 Conclusions

Two methods of calculating minimum distance between set-theoretically defined models have been devised and implemented. The first is a simple brute force approach which is useful to check the results obtained from the second, more sophisticated method. This second method calculates bounds on the minimum distance by using adaptive spatial sub-division; dividing the model more finely in areas where the closest two points are likely to be found.

The brute force method first divides the model and then calculates the distance between every pair of sub-model's boxes. The distance is found as the distance between the two closest boxes. This has computational efficiency of $O(mn)$, where m is the number of leaf boxes containing one object and n is the number of leaf boxes containing the other object. Despite its inefficiency, this method offers an important tool for checking the results obtained from using other, more sophisticated methods.

Empirical results have shown that the order of the highest order primitive affects the computational time in a linear fashion; this is likely to be because higher order primitives generally result in more complicated shapes which require more division. However, examining the computational efficiency of the adaptive division method using other metrics such as number of primitives has been found to be inconclusive. This is due to the nature of the binary trees used to store the divided model, which is such that large areas of the model containing many primitives may be ignored.

Experimental results from different methods of estimating the upper bound on the distance between two sub-models have been compared, and the more complicated methods do not tend to improve the overall efficiency of the method, as their increased computational overhead is not offset by the gain in computational efficiency due to the better estimations of the upper bound.

Chapter 6

Conclusions

This thesis has considered three areas of research in the field of set-theoretically defined geometry: the computation of the convex hull of a solid model, the labelling of the connected components of a solid model and the calculation of the minimum distance between solid models. Methods have been proposed and implemented that offer solutions to each of these points of research, and scope for further research in each area has been highlighted. The implementation of these methods has been carried out using the sVLIs [10] set-theoretic solid modelling kernel.

In the first part of the thesis two approaches have been proposed to find the convex hull of a set-theoretically defined model: a multi-dimensional approach, and a point-set based method. The multi-dimensional approach processes the set-theoretically defined model in a multi-dimensional space, finally projecting the convex hull of the model back down into the original space. Implementation of this method is left as an open problem. The point-set based method involves firstly generating a point set that represents the model, then finding the convex hull of the point set by using an existing convex-hull algorithm, before finally creating the set-theoretically defined convex hull from the resulting point set of the convex-hull algorithm. Three different ways of generating the points have been discussed, two of which result in the approximate convex hull for both non-polyhedral and polyhedral models, whilst the third finds the exact convex hull of polyhedral models only. The first of these methods creates convex hulls which are closer to the original shape of the model, and so would have applications in the field of computer vision and image recognition. The convex hulls generated by the second method, however, are guaranteed to contain the original model, therefore making it more applicable for applications such as collision detection. Lastly, the third point-generation method constructs exact convex hulls for polyhedral models, therefore making it the optimum method to use for such models.

The second area of research is connected component labelling, for which a method of labelling connected components within a model has been devised by adapting and extending Samet's quadtree algorithm [64] to work with binary trees of general dimensionality. These binary trees are created from set-theoretically defined models by using adaptive spatial sub-division. This method also enables the point connectivity query (see below) to be answered by the comparison of the labels of the parts of the sub-model that the two points lie in, although the path between the points will not actually be found. This labelling method has been implemented for three-dimensional models by using the sVLIs kernel modeller. The method has been specialised to handle polyhedral models analytically, in order to label their connected components faster than the general binary tree method. Both the general binary tree method and the method for polyhedral models have been tested on a variety of models, and each method labels the components correctly above a certain threshold of division resolution - this threshold varies depending on the choice of model. For polyhedral models however, using finer division resolutions for the specialised method does not result in a divided model which has a significantly greater size, so therefore this method can be used with a finer division resolution for all models, whilst only using a small amount of additional memory space.

In connected component labelling, the question of point connectivity has also been covered, namely whether or not two points in a set-theoretically defined solid model are connected by a path that passes entirely through solid parts of the model. In an effort to resolve this query, a line-segment intersection technique has been proposed, using a 'divide and conquer' approach to find a path connecting a series of points. Implementation in a multi-dimensional modeller to find the connectivity of two points results in a tree path between the points in a configuration space which corresponds to non-colliding movements of a model through the model's space. The implementation has shown this technique to be effective for model analysis in confirming that two points in solid parts of the model, or two points in air parts of the model, are connected.

The last area of research covers the calculation of the minimum distance between two set-theoretically defined solid models. For this, an algorithm has been proposed and implemented using the sVLIs kernel modeller which calculates the lower and upper bounds on the minimum distance. Depending upon the proposed application of the minimum-distance information, the algorithm can be used in different 'modes', for example, for the calculation of approximate bounds in a small amount of time, or for ascertaining more accurate bounds given more computational time. The algorithm has been found to give accurate results. Investigation into how certain properties of the models possibly affect the calculation time of the minimum distance was undertaken; these model properties being the number of primitives in the model, the algebraic complexity of the primitives, the 'crinkliness' of the model, and the distance between the objects. Of these, it was found that only the algebraic complexity of the primitives had any significant effect on the time taken to calculate the minimum distance bounds.

References

[1] S. G. Akl and G. T. Toussaint. Efficient convex hull algorithms for pattern recognition applications. In *Proceedings of the 4th International Joint Conference on Pattern Recognition*, pages 483–487, Kyoto, Japan, 1979.

[2] D. C. S. Allison and M. T. Noga. Computing the three-dimensional convex hull. *Computer Physics Communications*, 103(1):74–82, 1997.

[3] J. L. Bentley, M. G. Faust, and F. P. Preparata. Approximation algorithms for convex hulls. *Communications of the ACM*, 25(1):64–68, 1982.

[4] J. L. Bentley and M. I. Shamos. Divide-and-conquer in multi-dimensional space. In *Proceedings of the 8th ACM Annual Symposium on Theory of Computation*, pages 220–230, 1976.

[5] J. L. Bentley and M. I. Shamos. Divide and conquer for linear expected time. *Information Processing Letters 7, 2*, pages 87–91, 1978.

[6] J. E. Bobrow. Direct minimization approach for obtaining the distance between convex polyhedra. *International Journal of Robotics Research*, 8(3):65–76, 1989.

[7] J. D. Boissonnat, A. Cerezo, O. Devillers, J. Duquesne, and M. Yvinec. An algorithm for constructing the convex-hull of a set of spheres in dimension-D. *Computational Geometry - Theory and Applications*, 6(2):123–130, 1996.

[8] K. H. Borgwardt. Average complexity of a gift-wrapping algorithm for determining the convex-hull of randomly given points. *Discrete and Computational Geometry*, 17(1):79–109, 1997.

[9] A. Bowyer. Configuration space maps. The Svlis-M project. Dept. of Mechanical Engineering, University of Bath, England. http://www.bath.ac.uk/~ensab/G_mod/Svm/csm.html.

[10] A. Bowyer. *SVLIs —Introduction and User Manual.* Information Geometers, 2nd edition, 1995. http://www.bath.ac.uk/~ensab/ G_mod/Svlis.

[11] A. Bowyer, D. C. R. Eisenthal, D. Pidcock, and K. Wise. Configurations, constraints, and CSG. In *Proceedings of the 1st Korea-UK Workshop on Geometric modelling and Computer Graphics*, Seoul, Korea, April 2000.

[12] A. Bowyer, D. C. R. Eisenthal, and K. D. Wise. Pers. comm. The Svlis-M project. Dept. of Mechanical Engineering, University of Bath, England. http://www.bath.ac.uk/~ensab/G_mod.

[13] S. Cameron. Efficient intersection tests for objects defined constructively. *International Journal of Robotics Research*, 8(1):3, 1989.

[14] S. A. Cameron. Pers. comm. Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, England.

[15] S. A. Cameron and R. K. Culley. Determining the minimum translational distance between two convex polyhedra. In *Proceedings of the 1986 IEEE International Conference on Robotics and Automation (Conf. code 08325)*, pages 591–596, San Francisco, CA, USA, 1986.

[16] J. Canny. Computing roadmaps of general semi-algebraic sets. *Computer Journal*, 36(5):504–514, 1993.

[17] J. F. Canny. *The Complexity of Robot Motion Planning.* ACM Doctoral Dissertation Award. The MIT Press, 1988.

[18] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *Journal of the ACM*, 17(7):78–86, 1970.

[19] F. Chin and C. A. Wang. Optimal algorithms for the intersection and the minimum distance problems between planar polygons. *IEEE Transactions on Computers*, C-32:1203–1207, 1983.

[20] A. Datta and S. K. Parui. Dynamic neural net to compute convex hull. *Neurocomputing*, 10(4):375–384, 1996.

[21] A. M. Day and D. Tracey. Parallel implementations for determining the 2-D convex hull. *Concurrency Practice and Experience*, 10(6):p449–466, 1998.

[22] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Khokhar. Randomized parallel 3-D convex hull algorithm for coarse grained multicomputers. In

Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (Conf. Code 43717), pages 27–33, Santa Barbara, CA, US, July 1995. ACM, New York, NY, USA.

[23] M. B. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, 39(2):253–280, 1992.

[24] M. Dyer, J. Nash, and P. Dew. Optimal randomized planar convex hull algorithm with good empirical performance. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, (Conf. Code 43717)*, pages 21–26, Santa Barbara, CA, USA, July 1995. ACM, New York, NY, USA.

[25] H. Edelsbrunner, J. Vanleeuwen, T. Ottmann, and D. Wood. Computing the connected components of simple rectilinear geometrical objects in d-space. *R.A.I.R.O. Informatique Theorique-Theoretical Informatics*, 18(2):171–183, 1984.

[26] A. Ferreira, A. Rauchaplin, and S. Ueda. Scalable 2-D convex hull and triangulation algorithms for coarse grained multicomputers. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing (Conf. Code 44002)*, pages 561–568, San Antonio, TX, USA, October 1995. IEEE, Los Alamitos, CA, USA.

[27] E. Gasparraj and S. Anand. New three dimensional convex hull algorithm and it's applications in flatness evaluation. In *Proceedings of the 5th Industrial Engineering Research Conference (Conf. Code 46228)*, pages 239–244, Minneapolis, MN, USA, May 1996. IIE, Norcross, GA, USA.

[28] E. G. Gilbert and C. P. Foo. Computing the distance between general convex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation*, 6(1):53–61, 1990.

[29] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. Fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, 1988.

[30] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information processing letters, 1*, pages 132–133, 1972.

[31] P. J. Green and B. W. Silverman. Constructing the convex hull of a set of points in the plane. *Computer Journal*, 22(3):262–266, 1979.

[32] L. Guibas, D. Salesin, and J. Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. *Algorithmica (New York)*, 9(6):534–560, 1993.

[33] J. Heintz, M. F. Roy, and P. Solerno. Description of the connected components of a semialgebraic set in single exponential time. *Discrete and Computational Geometry*, 11(2):121–140, 1994.

[34] J. Hershberger and S. Suri. Applications of a semidynamic convex-hull algorithm. *BIT*, 32(2):249–267, 1992.

[35] F. J. Hill and G. R. Peterson. *Introduction to Switching Theory and Logical Design.* Wiley, New York, 1874.

[36] R. Hummel. Connected component labelling in image processing with mimd architectures. In *Intermediate-level image processing*, pages 101–127. Academic Press, New York, 1986.

[37] G. Hurteau and N. F. Stewart. Distance calculation for imminent collision indication in a robot system simulation. *Robotica*, 6:47–51, 1988.

[38] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters, 2*, pages 18–21, 1973.

[39] M. Kallay. Convex hull algorithms in higher dimensions. 1981.

[40] M. Kallay. Complexity of incremental convex hull algorithms in r^d. *Information Processing Letters*, 19(4):197, 1984.

[41] T. C. Kao and G. D. Knott. An efficient and numerically correct algorithm for the 2-d convex-hull problem. *BIT, 1990*, 30(2):311–331, 1990.

[42] Y. S. Kim. Recognition of form features using convex decomposition. *Computer Aided Design*, 24(9):461–476, 1992.

[43] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex-hull algorithm? *Siam Journal on Computing*, 15(1):287–299, 1986.

[44] A. Klinger and C. R. Dyer. Experiments in picture representation using regular decomposition. *Computer Graphics and Image Processing*, (5):68–105, 1976.

[45] A. Klinger and M. L. Rhodes. Organization and access of image data by areas. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 50–60, January 1979.

[46] D. T. Lee and F. P. Preparata. Computational geometry - a survey. *IEEE Transactions on Computers*, 33(12):1072–1101, 1984.

[47] Y. Leung, J. S. Zhang, and Z. B. Xu. Neural networks for convex hull computation. *IEEE Transactions on Neural Networks*, 8(3):601–611, 1997.

[48] M. C. Lin and J. Canny. A fast algorithm for incremental distance calculation. In *International Conference on Robotics and Automation*, pages 1008–1014, Sacramento, April 1991.

[49] M. A. Lopez and T. Ramakrishna. On computing connected components of line segments. *IEEE Transactions on Computers*, 44(4):597–, 1995.

[50] T. Lozano-Pérez. Spatial planning: a configuration space approach. *IEEE Transactions on Computers*, 32(2):108–119, 1983.

[51] R. Lumia. A new three-dimensional connected components algorithm. *Computer Vision, Graphics and Image Processing*, 23:207–217, August 1983.

[52] S. Meeran and A. Shafie. Optimum path planning using convex hull and local search heuristic algorithms. *Mechatronics*, 7(8):737–756, 1997.

[53] P. Milne. On the algorithms and implementation of a geometric algebra system. Technical Report 90-40, University of Bath Computer Science Department, 1990.

[54] D. Mollison. Conjecture on the spread of infection in two dimensions disproved. *Nature*, 240:467–468, 1972.

[55] S. Olariu, J. L. Schwing, and J. Zhang. Fast adaptive convex hull algorithm on two-dimensional processor arrays with a reconfigurable bus system. *Computer Systems Science and Engineering*, 10(3):131–137, 1995.

[56] B. Paden, A. Mees, and M. Fisher. Path planning using a Jacobian-based freespace generation algorithm. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1732–1737, 1989.

[57] D. Pidcock and A. Bowyer. Finding good bounds on the minimum distance between two set-theoretically defined geometric models. In *Proceedings of the*

CSG '98 Conference, Ammerdown, U.K., April 1998. Information Geometers Ltd.

[58] F. P. Preparata. Optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22(7):402–404, 1979.

[59] F. P. Preparata and S. J. Hong. Convex hulls of finite sets in two and three dimensions. *Communications of the ACM*, 20(2):87–93, 1977.

[60] F. P. Preparata and M. I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag New York Inc., 1985.

[61] W. E. Red. Minimum distances for robot task simulation. *Robotica*, 1(4):231–238, 1983.

[62] J. Rooney and P. Steadman, editors. *Computer Aided Design.* Pitman / Open University, 1987.

[63] A. Rosenfeld. *Picture Processing by Computer.* Academic Press, 111 Fifth Avenue, New York, UK edition, 1969.

[64] H. Samet. Connected component labelling using quadtrees. *Journal of the Association for Computing Machinery*, 28(3):487–501, 1981.

[65] H. Samet. Neighbor finding in images represented by octrees. *Computer Vision, Graphics and Image Processing*, 46(3):367–386, 1989.

[66] H. Samet and M. Tamminen. Computing geometric properties of images represented by linear quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(March), 1985.

[67] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, 1988.

[68] J. T. Schwartz. Finding the minimum distance between two convex polygons. *Information Processing Letters*, 13(4,5):168–170, 1981.

[69] T. Siméon. Planning collision-free trajectories by a configuration space approach. *Geometry & Robotics, in Springers Lecture Notes in Computer Science Series*, 391:116–132, 1988.

[70] E. Soisalonsoininen. On computing approximate convex hulls. *Information Processing Letters*, 16(3):121–126, 1983.

[71] I. Stojmenovic and E. Soisalonsoininen. A note on approximate convex hulls. *Information Processing Letters*, 22(2):55–56, 1986.

[72] K. Sugihara. Robust gift wrapping for the three-dimensional convex hull. *Journal of Computer and System Sciences*, 49(2):391–407, 1994.

[73] L. Thurfjell, E. Bengtsson, and B. Nordin. A new three-dimensional connected components labeling algorithm with simultaneous object feature extraction capability. *Computer Vision, Graphics, and Image Processing*, 54(4):357–364, July 1992.

[74] C. Turnbull and S. Cameron. Computing distances between nurbs-defined convex objects. *ICRA*, May 1998.

[75] I. D. Voiculescu. Implicit function algebra in set-theoretic geometric modelling. Technical report, Dept. of Mechanical Engineering, University of Bath, England, 1998. http://www.bath.ac.uk/~enpidv/Report.

[76] K. D. Wise. Pers. comm. Dept. of Mechanical Engineering, University of Bath. http://www.bath.ac.uk/~enskdw.

[77] K. D. Wise. *Computing Global C-space Maps using Multidimensional Set-theoretic Modelling.* PhD thesis, Dept. of Mechanical Engineering, University of Bath, England, 2000.

[78] J. R. Woodwark. Pers. comm. Information Geometers, Winchester, England. http://www.inge.com.

[79] J. R. Woodwark. *Computing Shape.* Butterworths, 1986.

[80] J. R. Woodwark and K. M. Quinlan. The derivation of graphics from volume models by recursive division of the object space. *Proceedings of the Computer Graphics Conference*, pages 335–343, August 1980.

[81] S. Zeghloul and P. Rambeaud. A direct minimization approach for obtaining the distance between convex polyhedra - comment. *International Journal of Robotics Research*, 11(5):499–501, 1992.

[82] S. Zeghloul and P. Rambeaud. A fast algorithm for distance calculation between convex objects using the optimization approach. *Robotica*, 14(4):355–363, 1996.

[83] J. Zhou, X. Deng, and P. Dymond. 2-D parallel convex hull algorithm with optimal communication phases. In *Proceedings of the 11th International Parallel Processing Symposium (Conf. Code 46339)*, pages 596–602, Geneva, Switzerland, April 1997. IEEE, Los Alamitos, CA, USA.