COLLISION DETECTION BY FOUR-DIMENSIONAL INTERSECTION TESTING

by

Stephen Cameron

Collision Detection by Four-Dimensional Intersection Testing[*]

Stephen Cameron

Abstract

The collision detection problem is easily stated: "Given two objects and desired motions, decide whether the objects will come into collision over a given time span". The solution of this problem is useful, both in robotics and other problem domains. We describe a method for solving collision detection that involves transforming the problem into an intersection detection problem over space-time. We give the theoretical basis for the solution, and describe an efficient implementation based on describing the objects and motions constructively. We also consider the related problems of describing the collision region, and of detecting collisions when there are a more than two moving objects.

---

1 Introduction

The collision detection problem may be stated as: "Given two objects and *desired* motions, decide whether the objects will come into collision over a given time span". Solving this problem is useful in its own right, and the solution can also be used as parts of algorithms that try to *generate* collision-free paths. We have been especially interested in solving the collision-detection problem for robotics, but the work reported will also be of interest in other problem domains, such as VLSI and electronic circuit layout, "Cloth-cutting", bin-packing, assembly planning, and driving numerically-controlled machines. Previous work on this problem has mainly arisen from two sources. One source has been the proliferation of CAD descriptions of shapes, and the desire to do more design on the computer (e.g., [Boy79, Mey81, Mye81, CK86]). Another source has been from algorithm design in its own right, which emphasises the design of algorithms with low computational complexity. Our work is rooted firmly in the former camp, but with emphasis on the production of efficient algorithms.

There are many different algorithms for collision detection. As argued in [Cam85], one class of algorithms is conceptually the simplest: we choose a number of times, $\{t_i\}$, within our time-span of interest, and perform a (static) interference test at each t_i. This algorithm has many advantages: it is relatively simple, it is not necessary to derive a closed-form for the motion (access to a sampling function will suffice), and it gives good operator feedback when used as part of simulation. However, the algorithm is not perfect, and, in particular, it does not work well if objects come into contact. Another method is to compute the volume swept out by the objects over their motions, and to declare a collision if these swept volumes intersect. Again this method is intuitive but, as we shall see, the method described in this paper is effectively better (except in some "special cases").

[Can86] describes yet another method for collision detection, in which the problem is transformed into detecting collisions for a point in *configuration space* [LP83]. Effectively Canny considers intersection detection between a line and six-dimensional configuration space obstacles, using algebraic techniques to find the intersection regions. Configuration space is normally associated with solutions of the collision *avoidance* problem (e.g., [LP87, Don87, Can88]). We believe that collision detection is worthy of separate study as it can often be solved far faster than collision avoidance; also, many collision avoidance schemes require a collision detector to be run first to generate information about the collision region (e.g., [Mye81]).

In §2 we give the formal basis for four-dimensional intersection testing. To visualise the process, we may imagine an analogue, whereby we perform

collision-detection in a *two*-dimensional world [Abb52]. Imagine the two polygonal objects shown in figure 1(a) as starting from the positions shown and having the velocities arrowed. Now imagine that the two-dimensional universe that they inhabit is, in fact, the floor of a lift[1], which is moving vertically upwards with some constant velocity. Then the polygons will sweep out prisms in three-dimensional space, as shown in figure 1(b). We can think of the vertical dimension in this case as being a *time* dimension; taking a particular horizontal slice of these prisms gives the positions of the polygons at the corresponding time. Then, as we shall show, the polygons collide if and only if the prisms intersect. Figure 1(c) shows the union of the two prisms, and figure 1(d) shows their (non-null) intersection.

§3 describes how this problem transformation is performed within a geometric modelling system called ROBMOD [CA88]. §4 gives the meat of the implementation, which is based on the more general routine described in [Cam89]; examples of the routine in action are given in §5. The routine has a natural extension to tackle the collision detection problem when there are many possible pairs of objects that could collide; this is outlined in §6. Further extensions, and connections with work on other problems, are described in §7.

2 Mathematical Basis

We regard an object as being defined by a point set. (This is equivalent to assuming that we know exactly where objects are, and what their shape is.) Given an object O, we assume the existence of a *location function*, Λ, which is a function that tells us where the object is at a given time. In particular, Λ takes a time t and returns a transformation $\Lambda(t)$ which tells us how to move O into its position at time t, and so at this time O occupies the point set

$$\{x \mid (\exists y)\, y \in O \text{ and } x = \Lambda(t)(y)\}$$

which we normally write as $\Lambda(t)(O)$. As an example, an object which is at its rest position at time 0 and has a constant velocity v has a location function that moves the point x to the point $x + vt$ at time t. We will only be concerned with rigid-body motions, but we note that most of the motions made by "normal" materials, including elastic deformations and fluid flows, can (in principle) be described by *invertible, continuous* location functions, as matter is not lost during the transformation.
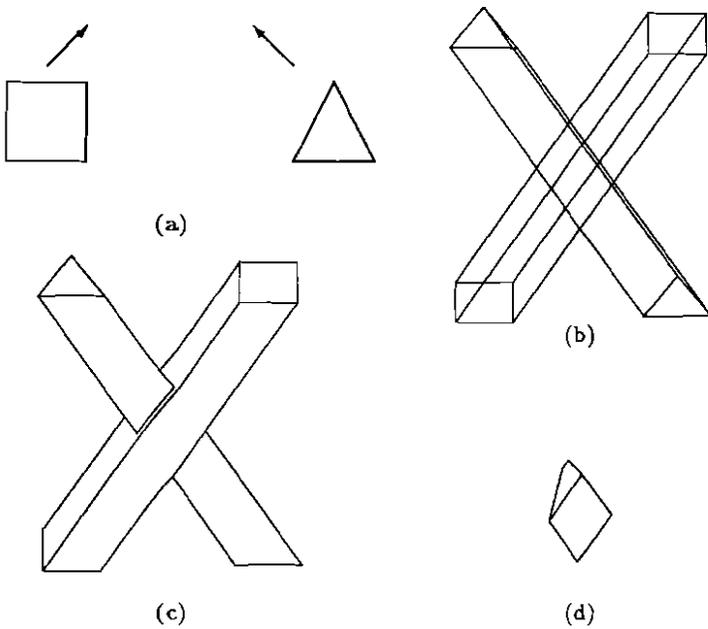
---

[1] *aka.* "elevator".

Figure 1: Example of extruding from two-dimensions into three.

2.1 Extrusions

The above definitions give us enough structure to formally define the extrusion operation that we introduced informally in §1. Given an object O and corresponding location function Λ, we define the extrusion operator **Ex** by

$$\mathbf{Ex}(\Lambda, O) \;\; = \;\; \{(\mathbf{x}, t) \,|\, \mathbf{x} \in \Lambda(t)(O)\} \tag{1}$$

An equivalent definition (for invertible transformations) is

$$\mathbf{Ex}(\Lambda, O) \;\; = \;\; \{(\mathbf{x}, t) \,|\, \Lambda(t)^{-1}(\mathbf{x}) \in O\} \tag{2}$$

Then we note that object A (with location function Λ_A) collides with object B (with location function Λ_B) if and only if

$$\exists (\mathbf{x}, t) \;\; \mathbf{x} \in \Lambda_A(t)(A) \text{ and } \mathbf{x} \in \Lambda_B(t)(B) \Leftrightarrow$$
$$\mathbf{Ex}(\Lambda_A, A) \cap \mathbf{Ex}(\Lambda_B, B) \neq \emptyset$$

i.e., two objects collide if and only if their extrusions intersect.

Thus extrusions give us a mathematical framework for considering the collision detection problem. However, these definitions do not tell us how to *construct* extrusions. [ER83] considers the extrusion of collections of simple shapes, namely spheres. [ST85, FS89] use a hierarchal description called a *bintree*[2] to describe space-time; the bintree is constructed using an algorithm similar to that described in §4.1, but always continues division until the space-time region is full or empty. [Gla88] uses space-time to perform ray-tracing of moving objects for animation, for which only samples of the extrusion are required. [JP88] give a construction for the boundary of extrusions when the location functions correspond to linear velocities and the objects are polyhedral, but in general the boundary of an extrusion may be quite complex.

In our work we have used *constructive solid geometry* (CSG) as the method for describing three-dimensional shapes, whereby a shape is given as a set-combination of simple shapes—this is a common description method for solid models [RV82]. Effectively, a CSG description is equivalent to a Boolean function F of a number of simple objects P_i, so that the object is given by

$$F(P_1, P_2, \ldots, P_m)$$

where F is obtained by use of the set operations of union ($\cup$), intersection ($\cap$) and difference ($/$). Given such an F we may derive the function $\hat{F}$, obtained from F by replacing each three-dimensional set operation by the

---

[2]Similar to a quadtree [Sam84]

corresponding four-dimensional set operation[3]. Then the following *distri-bution theorem* gives us a way of simplifying the construction of extrusions:

◇ Distribution Theorem

The extrusion operation distributes over the set operations; that is

$$\mathbf{Ex}(\Lambda, F(P_1, P_2, \ldots, P_m)) =$$
$$\hat{F}(\mathbf{Ex}(\Lambda, P_1), \mathbf{Ex}(\Lambda, P_2), \ldots, \mathbf{Ex}(\Lambda, P_m))$$

Proof (for the standard set operations) Let $\check{F}$ denote the logical formula derived from a set formula F in the normal way (by replacing $\cup$ by $\vee$, etc.). Then

$$\begin{aligned}
(\mathbf{x}, t) &\in \mathbf{Ex}(\Lambda, F(P_1, P_2, \ldots, P_m)) \\
&\Leftrightarrow \quad \Lambda(t)^{-1}(\mathbf{x}) \in F(P_1, P_2, \ldots, P_m) \\
&\Leftrightarrow \quad \check{F}(\Lambda(t)^{-1}(\mathbf{x}) \in P_1, \ldots, \Lambda(t)^{-1}(\mathbf{x}) \in P_m) \\
&\Leftrightarrow \quad \check{F}((\mathbf{x}, t) \in \mathbf{Ex}(\Lambda, P_1), \ldots, (\mathbf{x}, t) \in \mathbf{Ex}(\Lambda, P_m)) \\
&\Leftrightarrow \quad (\mathbf{x}, t) \in \hat{F}(\mathbf{Ex}(\Lambda, P_1), \mathbf{Ex}(\Lambda, P_2) \ldots, \mathbf{Ex}(\Lambda, P_m))
\end{aligned}$$

as required.

The distribution theorem simplifies the construction of extrusions as it is often easy to write down the form of an extrusion of a simple shape P_i. In particular, if each P_i is a *half-space*:

$$P_i = \{\, \mathbf{x} \mid p_i(\mathbf{x}) \leq 0\,\} \quad \text{for some function } p_i$$

then the extruded half-space is another half-space:

$$\mathbf{Ex}(\Lambda, P_i) = \{\, (\mathbf{x}, t) \mid p_i \circ \Lambda(t)^{-1}(\mathbf{x}) \leq 0\} \tag{3}$$

where ∘ denotes functional composition. One specific case that is of interest is when each half-space of the object is linear, and moving with constant linear velocity. So if P corresponds to the half-space $p(\mathbf{x}) = \mathbf{n}.\mathbf{x} + d$ (where . is the scalar product operator) and Λ is the location function corresponding to the identity transform at time 0 and a constant velocity $\mathbf{v}$, we find that $\mathbf{Ex}(\Lambda, P)$ corresponds to the (four-dimensional) half-space

$$\hat{p}(\mathbf{x}, t) = \mathbf{n}.\mathbf{x} - \mathbf{n}.\mathbf{v}t + d \tag{4}$$

---

[3] Essentially F and $\hat{F}$ look the same; they are distinguished for reasons of pedantry.

We use the combination of (3) and the distribution theorem to model the extrusions of objects. If we have two extrusions, say

$$\hat{F}(\hat{P}_1, \hat{P}_2, \ldots, \hat{P}_m) \text{ and } \hat{G}(\hat{Q}_1, \hat{Q}_2, \ldots, \hat{Q}_n),$$

where $\hat{P}_1$, etc., denote extruded primitives, then the objects collide if and only if the extrusions intersect; that is, if

$$\hat{H}(\hat{P}_1, \hat{P}_2, \ldots, \hat{P}_m, \hat{Q}_1, \hat{Q}_2, \ldots, \hat{Q}_n) \neq \emptyset$$

where

$$\hat{H}(\hat{P}_1, \ldots, \hat{P}_m, \hat{Q}_1, \hat{Q}_2, \ldots, \hat{Q}_n) = \hat{F}(\hat{P}_1, \ldots, \hat{P}_m) \cap \hat{G}(\hat{Q}_1, \ldots, \hat{Q}_n)$$

So we have transformed the collision detection problem into one of detecting whether any (four-dimensional) point satisfies a region given by a set theoretic formula.

2.2 Mathematical Niceties

The standard set operations are *not* generally used in geometric modelling practise, as it is possible to construct non-three-dimensional objects with them. Instead the (closed) *regularised* set operations are used; these are equivalent to performing a standard set operation, followed by taking the closure of the interior of the resultant set. (Informally: we perform the set operation, and stretch a tight skin over the resultant set.) These operations can be shown to form a Boolean algebra [TR80], and the main problem in using them is that we have to be a careful when considering the boundaries of sets formed. However, the following result can be shown [Cam84]:

◇ Distribution Theorem (for regularised sets)

The extrusion operator **Ex** distributes over the (closed) regularised set operations.

Proof Here we will just give an outline proof of the theorem; details are to be found in [Cam84].

By the same argument as was used for the standard set operations, we find that

$$(\mathbf{x}, t) \in \mathbf{reg}(\mathbf{Ex}(\Lambda, F(P_1, \ldots, P_m)))$$
$$\Leftrightarrow (\mathbf{x}, t) \in \hat{F}(\mathbf{Ex}(\Lambda, P_1), \ldots, \mathbf{Ex}(\Lambda, P_m))$$

where **reg** is the (four-dimensional) regularisation operation. Thus the result will follow if $Ex(\Lambda, F(P_1, \ldots, P_m))$ is a regular set. But $F(P_1, \ldots, P_m)$ is regular (in three dimensions), and as Λ and its inverse are both continuous bijections we can show that $Ex(\Lambda, F(P_1, \ldots, P_m))$ will be regular, provided that the time span of interest is regular in one dimension, i.e., as long as the time domain forms a closed set. To see that this is a reasonable requirement, consider the extrusions of the two-dimensional objects in figure 1; if the extrusion is carried out over an open time interval, the 'tops' and 'bottoms' of the extrusions will be open, and so the extrusions would not be regular.

Another property of "well-behaved" geometric models is that they be *triangulable* [Req77]; effectively, that their boundaries can be described by a finite number of elements. For the work reported here triangulability follows from the finiteness of the CSG descriptions; however it is interesting to note that triangulability is preserved under extrusion for the standard location functions [Cam84].

2.3   Connection with Sweeping

As mentioned in §1, a common way of performing collision detection is to compute the volume swept out by each object, and test these swept volumes for interference. Sweeping can be formalised by introducing the operator **Sw**, with

$$Sw(\Lambda, O) = \{ x \mid \exists(y, t) \; x = \Lambda(t)(y) \}$$

Comparing this equation with (1) we note that sweeping is equivalent to extrusion into space-time followed by a projection operation back into the original space, and thus that, functionally, sweeping is more complex than extrusion. It also explains why sweeping two moving objects and testing for interference is not a sufficient test for collisions between the objects; the objects might occupy the the same space at different times, but this temporal information is suppressed by the sweeping operation. Sweeping can be made a sufficient test by considering the relative motions of two objects. However, such relative motions may be complex, and if there are many objects moving we may have to consider many pairs of relative motions. Using extrusion to solve the many-pair problem is more promising, as explained in §6.

Given this added in-built complexity of sweeping over extrusion, it is interesting to speculate on the popularity of the sweeping method. We postulate two reasons. Firstly, for some shapes and some motions, the swept volume has a particularly nice form. For example, in [dPBB83] spheres are rotated and translated to form volumes that can be modelled using toroids, cylinders and spheres. Thus for these cases a fairly conventional solid modeller can be used. The second reason is a lack of familiarity with

the mathematics involved. (We hope that this paper might help to alleviate such fears.)

3   Implementation

In our implementation, which is part of a solid modelling system called ROB-MOD [CA88], shapes are entered as expressions, that describe the shapes in terms of parameterised simple shapes (such as blocks and cylinders), together with rigid-body transformations, joint parameters (for mechanisms), and set operations. These descriptions are read by a parser that converts them into a tree structure, whose leaf nodes reference instances of simple shapes, and whose branch nodes either represent transformations or binary set operations. For simplicity we may imagine these trees to be equivalent to trees without the transformation nodes, i.e., whose branch nodes denote only binary set operations, and whose leaf nodes include the appropriate transformation together with the simple shape.

We have used the construction paradigm to denote location functions also. A ROBMOD expression of type motion is made up of a chain of primitive motion commands, together with the times for which each is applicable. For example, the expression

```
rest until 0 vel 1 2 3 until 5 vel 0 1 1 until 10 rest
```

denotes a motion that is at rest until time 0 and after time 10, the velocity $(1, 2, 3)$ between times 0 and 5, and the velocity $(0, 1, 1)$ between times 5 and 10. This effectively gives us the derivative of the location function; to fix a particular function we specify that a motion corresponds to the identity transformation at time 0. This syntax was used for convenience only; other syntax could easily be used in its place (such as specifying via points).

To attach a motion expression to a given shape, we make a ROBMOD worm expression[4] by connecting a shape expression to a motion expression. The collision detection function is given two worms as input, together with a time-bound over which to test for collisions. (The time-bound is not strictly necessary, as will be explained in §4.3.) In turn, each worm is presented internally as a binary shape tree, together with a list of primitive motion components for that shape. These inputs are further converted into a single binary tree whose branch nodes are set operations and whose leaf nodes correspond to four-dimensional half-spaces. The rest of this section gives the procedure for generating this tree, and §4 gives the procedure for testing

---

[4]So called because we can imagine the corresponding extrusions as "worms" in space-time

whether this tree corresponds to the null set (and thus whether the objects collide within the given time bounds).

The procedure for deriving the tree, given a *single* motion component, is straightforward:

1. The leaf nodes in the shape tree correspond to complete simple shapes. We first rewrite each simple shape into an appropriate combination of half-spaces—for example, a block is replaced by the intersection of six linear half-spaces.

2. The shape tree is now extruded—effectively, by extruding each primitive (using (3)) and replacing each three-dimensional set operation by its four-dimensional version. (In fact, in ROBMOD the second operation is a null step, as there is a single set operation node, regardless of the dimensionality.)

3. This gives us an extrusion defined for all time. To limit the time to the time span of interest, say $t_l \leq t \leq t_h$, we intersect this extruded shape tree with the intersection of the two linear half-spaces $-t \leq -t_l$ and $t \leq t_h$.

Example Consider the block given by $-1 \leq x, y, z \leq 1$ moving with velocity $(1,1,1)$ for $0 \leq t \leq 10$, and at rest otherwise. Then the complete extrusion is given by the union of

1. The intersection of the seven half-spaces $-1 \leq x, y, z \leq 1$ and $t \leq 0$.

2. The intersection of the eight half-spaces $t - 1 \leq x, y, z \leq t + 1$, $t \geq 0$ and $t \leq 10$.

3. The intersection of the seven half-spaces $9 \leq x, y, z \leq 11$ and $t \geq 10$.

To construct a CSG description with multiple time components, say $t_1 < t_2 < \ldots < t_n$, we find the extrusion over each component separately, bound the tree between $\max(t_l, t_i)$ and $\min(t_h, t_{i+1})$ $(1 \leq i < n)$, and then take the set union of the extrusion trees to get the total extrusion. This gives us a binary tree that completely specifies each worm; these are then intersected (symbolically) to represent the entire region of space-time for which the objects overlap. (An alternative approach is to identify the time spans over which both objects have constant velocities, and to run the collision detection process separately for each time span. This is the approach used in §4.3.)

In our current ROBMOD implementation we restrict ourselves to generating only linear four-dimensional half-spaces. This is done simply by considering only polyhedral approximations to shapes, moving through motions which are composed of linear velocity segments. Thus we actually rewrite, say, a cylinder as the intersection of a number of (three-dimensional) linear half-spaces, and extrude all the half-spaces using (4).

4 Null Set Detection

We now have a four-dimensional intersection set, specified by a Boolean tree, and we want to see whether the set is empty. This can be regarded as a set satisfiability problem: does there exist a (four-dimensional) point that satisfies the set given by this Boolean formula? Several techniques exist for solving the intersection problem in three or fewer dimensions, and many of these techniques are amenable to tackling the four-dimensional problem. We will follow the general approach detailed in [Cam89], giving the modifications required for our particular geometric domain. Effectively, the algorithm is split into three stages, which operate in cascade to provide an efficient solution to the problem. These stages are:

1. A pre-processing stage, based on reasoning about approximations to subtrees. This stage is called the *S-bound preprocessing* stage.

2. A divide-and-conquer stage, whereby the problem is dynamically split into a number of simpler problems to reduce the computational complexity.

3. A generate-and-test stage, at which the exact geometry of the problem is considered.

The purpose of the cascade is to reduce the overall time cost of the algorithm, by using relatively cheap processing to solve the 'easy' parts of the problem and only passing onto the further stages the parts that are still in doubt. Here, a "part" means a rectangular[5] region of space-time in which we search for a point in the intersection set. Note that, for simplicity, we have not implemented the reduudancy-based routine described in [Cam89].

In order to follow the development of this algorithm, and to improve the presentation, we describe the S-bound preprocessing stage last.

---

[5]We use 'rectangular' to imply a product set of closed intervals, i.e., an aligned rectangle in two dimensions, a box in three, etc.

4.1 Divide-and-Conquer

The input to this stage is a CSG description, plus a rectangular region of space-time within which to search for evidence of non-nullity. (Finding such evidence implies that the extrusions overlap and so that the objects collide.) In our original implementation a bounding region of space was computed for each object by enclosing the object at every one of the points in time at which the velocity changes; the space-time region was then generated by intersecting the space regions for the two objects, and adding the time bounds given to the clash function. In the current implementation the space-time region is given directly by the S-bound preprocessing step (as described in §4.3).

This region and the intersection tree could be passed straight to the routine given in §4.2, but for reasons of computational efficiency we interpose a divide-and-conquer stage, which replaces our single problem by a number of smaller problems. The mechanism involved is discussed in detail in [Cam89], and we only give brief details here.

1. Given a region of space-time R, and a tree, T, we measure the complexity of T, and decide whether to continue to the generate-and-test routine (§4.2), or to divide the problem up.

2. To divide the problem we split the region R into a number of subregions $\{R_i\}$, with the subregions covering R. Then, for each R_i, we make a *simplified copy* of the intersection tree T_i, using the technique discussed below. The region/tree pairs $\langle R_i, T_i \rangle$ are then recursively evaluated (step 1).

3. The entire problem terminates whenever any subproblem discovers that the intersection set is non-null, or when all the subproblems have reached the generate-and-test stage.

Note that the space requirement of this process is proportional to the maximum depth of subdivision, and not to the total number of regions examined.

As each of our regions are aligned, rectangular boxes (in four-dimensions), then a simple strategy for splitting the regions is to split each box into sixteen parts by bisection along each coordinate axis. This is, in fact, the strategy that we have adopted, as it seems to work well; however a number of heuristics could be invoked to try to balance the size of the subproblems generated; [Woo86] gives examples of such heuristics in a three-dimensional situation.

The simplification strategy is based on the observation that if the boundary of a half-space does not pass through a region, then the corresponding

leaf can be removed. For our convex half-spaces and convex polyhedral regions, we can check whether the boundary intersects the region simply by computing the half-space function at the region extreme points. It is worth noting that we may often discover that a region simplifies to a null region, or a completely full region (proving non-nullity without having to consider the boundary intersections), or a region with only one or two half-space boundaries passing through it. In the latter case an efficient closed-form solutions exists, namely by treating the simplified tree as defining a formula of the propositional logic, and testing whether it is a contradiction.

Example A sphere of radius 4 has centre at $(5,5,5)$ at time 0, and moves with velocity $(1,1,1)$. A cube of sides 4 is centred at $(44,54,5)$ at time 0, and moves with velocity $(0,0,1)$. To test whether any collision occurs in $0 \le x, y, z, t \le 64$ we consider the intersection of the 7 extruded half-spaces

$$(x - 5 - t)^2 + (y - 5 - t)^2 + (z - 5 - t)^2 \le 16$$
$$42 \le x \le 46$$
$$52 \le y \le 56$$
$$3 \le z - t \le 7$$

The division mechanism quickly decides that only the space-time that is bounded by $40 < x < 48$, $52 < y < 56$, $44 < z < 52$ and $42 < t < 46$ contains any points of interest, and goes on to pass 8 regions of width 4 to the next stage for further investigation. This means that only approximately 0.01% of the original hypervolume is explored in detail.

Calculation of the computational complexity of this process is difficult, as the worst-case analysis is, experimentally, extremely pessimistic, and it is difficult to characterise a set of more realistic cases to give a measure of the expected complexity; however our analysis does suggest that the expected complexity is not worse than $O(n^2)$ [Cam89]. We can also apply some heuristics to speed up the process, such as relating our measure of 'complexity' of a tree to the size of the region. In our implementation we measure the complexity of a tree by its number of leaf nodes, and decide to 'conquer' instead of 'divide' if the complexity is smaller than $\kappa(d)$, where d is the number of division steps already performed. In our implementation we use $\kappa(d) = 2d + 6$ $(0 \le d \le 6)$. If the region size becomes very small we assume that the intersection hypervolume is so small that it can be ignored. In practise, this has never happened.

4.2 Generate–and–test

The generate-and-test routine is our general routine that checks for nullity. More sophisticated routines can be devised (e.g., [RV89]), but as the divide-and-conquer mechanism ensures that the problems given to our routine are bounded in size, we have chosen to go for simplicity, and follow the approach given in [Cam89]. This involves generating a sufficient set of test points (in space-time) and checking these points to see if any is inside the intersection set. To generate the point set, we go though a loop:

1. For every *triple* of half-spaces referenced by the tree, find their intersection. In the general case, this will be a *line* through space-time.

2. Intersect every line with every half-space. (We do not, of course, need to intersect with any of the triple from which this line was formed.) This gives a number of potential edge segments; if the intersection polytope is non-null, some of these edge segments will lie inside or on the intersection set.

3. For every edge segment, classify the mid-point.

This algorithm thus requires $O(n^4)$ point classifications. In a non-regularised set system, it would be sufficient to classify a point by evaluating the half-space functions at the point, and combine the Boolean truth values using $\wedge$ where we see set intersection, etc. Classification in this case is a linear time process, and so the total complexity of this stage is $O(n^5)$. However, in a regularised system we have to take the *neighbourhood* of the points into account[6]. Our choice of points to test—the mid-points of potential edges— is significant here, as we can then take a cross-section to the line at the test point. This reduces the problem to evaluating the intersection of three planes in three-dimensions, which is isomorphic to the problem of classifying a vertex in three-dimensions. In turn, this can be solved by considering the edges surrounding the vertex (which are the intersections of pairs of the original triple of half-space boundaries, together with the cross-section hypersurface), and using neighbourhood classification techniques directly on these. (Compare this with classifying an edge in three dimensions, by taking a cross-section perpendicular to the edge to reduce it to a two-dimensional classification problem.) Eventually the classification problem is reduced to testing a number of points, each of which can be tested using the logic formula approach above. Details are given in appendix A.

---

[6]Using regularised sets is essential if we wish to deal reliably with objects in contact.

4.3 S-bounds

The test for a null intersection given above works, and works well, but it is dumb in the way that it computes the initial space-time bound to consider. To illustrate this, imagine a pair of unit cubes, aligned with some world coordinate axes, with the first cube starting at the origin and moving with velocity $(1,1,1)$ for a length of time T, and the second starting at $(2,0,0)$ and moving with the same velocity—then the hypervolume considered will be $\Theta(T^4)$, whereas in a coordinate frame moving with velocity $(1,1,1)$ both cubes would be fixed, and the hypervolume considered would be $\Theta(T)$. In practise this is not too much of a drawback, as the divide-and-conquer algorithm would quickly prove large regions of space-time null, as they would be entirely outside one or other of the extrusions. However, S-bounds provide a way of focusing the attention of the algorithm; they also help to remove so-called redundant primitives from consideration [Cam89].

4.3.1 Overview of S-bounds

The binary tree representing the intersection set contains information about the relative constraints between the half-spaces due to the root node of the tree, and thus the relative constraints between subtrees. S-bounds give us a way of organising these constraints, so that we can quickly reason about which parts of the tree are mutually contradictive. S-bounds are described in detail in [Cam89]. An S-bound system is defined by a class of bounds, together with two operators $\sqcap$ and $\sqcup$. The bounds are subsets of space—in this case $\Re^4$—that are chosen to be easily described and manipulated. The operators must satisfy the rules:

$$A \sqcap B \supseteq A \cap B \qquad\qquad A \sqcup B \supseteq A \cup B$$

for all bounds A and B. ROBMOD uses rectangular boxes, aligned with the world coordinate system, as three-dimensional S-bounds (3DSBs), and then the operators are given by $A \sqcap B = A \cap B$, and $A \sqcup B$ is the smallest aligned box that contains $A \cup B$. Both of these operators can be implemented in unit time by simply taking the maximum and minimum of pairs of coordinates that define the corners of the box. Given a tree, an initial set of bounds is generated by setting the bounds at the leaf nodes to be supersets of the relevant primitive shapes, and Ω (the universal set) elsewhere. Such a set of bounds has the S-bounds property, namely that the set given by each subtree need not be evaluated outside of its appropriate bound. The real power of S-bounds lies in the fact that we can then rewrite the bound set using the set of rewrite rules in figure 2 to get a new, smaller set of bounds with the

---

Let T denote a general node of the tree, $L(T)$ its left child, $R(T)$ its right child, $P(T)$ its parent, and $\beta(T)$ its bound. Then we have the following rules:

Upward Rule: If T is a branch node, set $\beta(T) \leftarrow \beta(T) \cap S$ where

$$S = \left\{ \begin{array}{lll} \beta(L(T)) & \sqcap & \beta(R(T)) \\ \beta(L(T)) & \sqcup & \beta(R(T)) \\ \beta(L(T)) & & \end{array} \right\} \text{ if } T \text{ is a } \left\{ \begin{array}{l} \cap \\ \cup \\ / \end{array} \right\} \text{ node}$$

Downward Rule: If T is not the root node, set $\beta(T) \leftarrow \beta(T) \cap \beta(P(T))$
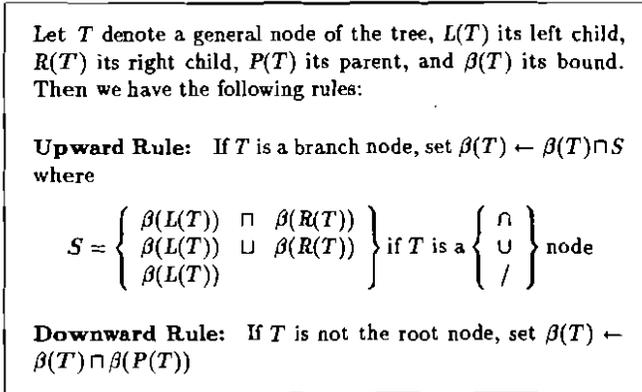

Figure 2: Upward and Downward rules for S-bounds.

S-bound property, where the Upward rule is first applied in a bottom-up manner throughout the tree, followed by the Downward rule in a top-down manner, and repeating. As shown in [Cam89], this procedure converges quickly for three-dimensional intersection detection problems, and leads to significant computational savings as we can often demonstrate that entire subtrees can be replaced by the null set, and thus need not be explored in detail.

4.3.2 S-bounds in Four Dimensions

When we discussed the problem with the standard divide-and-conquer algorithm we mentioned that the hypervolume to be considered can grow large if we bound the space relative to a moving frame. For the same reason, simply extending S-bounds to be rectangular regions of space-time is not as efficient as it might be. Thus we have decided to use a slightly more complicated S-bounds system for our four-dimensional intersection detection work, by choosing S-bounds that more exactly bound the extrusions.

Formally, our four-dimensional S-bounds (4DSBs) consist of the union of a number of convex polytopes in space-time, with the polytopes not overlapping in time. In particular, we split the problem up along the time dimension into a number of time spans, $[t_i, t_{i+1}]$, so that both objects are moving with constant velocities over each time span. (Thus if the objects

have m and n motion components, there will be at most $m + n$ time spans to consider.) Further, we choose the operators $\sqcap$ and $\sqcup$ so that the 4DSBs have a relatively simple form; each 4DSB is the extrusion of a 3DSB with the same motion as the corresponding object, except at the root node, where the 4DSB is the unevaluated intersection of the 4DSBs of its children. To see why this permits simple combination operations, consider two 4DSBs of this form, namely $\langle \Lambda, \beta_1 \rangle$ and $\langle \Lambda, \beta_2 \rangle$. Then we can see (by considering the spatial and temporal dimensions separately) that $\langle \Lambda, \beta_1 \rangle \square \langle \Lambda, \beta_2 \rangle \equiv \langle \Lambda, \beta_1 \square \beta_2 \rangle$ gives a suitable definition of the four-dimensional operators (where $\square$ is one of $\sqcap$ or $\sqcup$), using the standard three-dimensional aligned box operators. So, within the subtrees for each object, we can effectively use only the three-dimensional combination operators, and ignore the motions.

Matters are only slightly more complicated at the root node. We need to be able to intersect two rectangular regions of space-time moving with arbitrary linear velocities, and express the result as the intersection of two new rectangular regions of space-time, each moving with the same velocity as before. (We do not have to consider a $\sqcup$ operation here, as the root node is always an intersection node for collision detection.) We have computed a closed-form solution for this problem, which is detailed in appendix B. Note that when we consider this root node we may (and often do) generate a smaller time-bound than that originally given. In terms of the example of figure 1, this would be equivalent to placing a bounding rectangle around the triangle, and solving *exactly* for the space-time in which the rectangle bound and the square overlap. This by itself is not sufficient to prove that the triangle and the square overlap, but it does limit the search space for our divide-and-conquer algorithm.

Example Consider the example from §4.1. The original ROBMOD bounding procedure considers a space-time region of dimensions $4 \times 4 \times 68 \times 64$. Applying the closed-form solution to rectangular S-bounds in this case gives a space-time region of dimensions $2 \times 2 \times 4 \times 2$ instead!

5 Examples

Figure 3(a) shows a snapshot of a pair of composite objects, which are under motions that cause a collision. The two sets of objects are an autonomous vehicle, which is carrying a palleted load and is moving straight forward, and a line of trays, two of which are carrying loads and which are moving in a direction perpendicular to the motion of the vehicle (supported from an invisible overhead rail). In terms of geometric complexity,
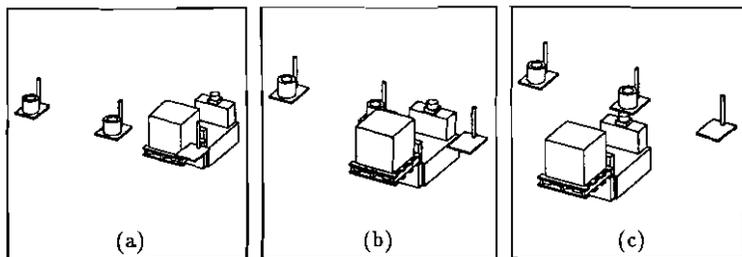
Figure 3: Vehicle Moving in a Straight Line

the composite objects are described by 19 and 10 primitive shapes in the
CSG descriptions, which require about 150 linear half-spaces to describe.
Figures 3(b) and (c) show two later snapshots, with the former showing a
collision between a loaded tray and the load of the vehicle. The collision de-
tection routine was asked to search for clashes over a time span of length 20;
the S-bound stage correctly identified a subspan of length 1.33 as being of
interest, and found a witness to the collision (a point in space-time at which
the collision was occurring) in 0.5s of CPU time (on a SUN 3/260 without a
floating point accelerator). To illustrate the usefulness of the S-bound stage
here, note that only 5 of the 29 original primitives survived the S-bound
stage after 2 Up/Down passes[7], reducing the hypervolume to be considered
by a factor of 20. (These figures are for illustration only; in practise the
regions discarded by the S-bound stage would have otherwise been quickly
discarded by the division stage.)

Instead of terminating when any point of collision is found, the routine
can also be asked to find an earliest witness (a point when the collision
starts). This is done by ordering the division stage of the routine so that
the 'earliest' regions are examined first, and only terminating when the
routine is sure that the earliest point has not been missed. As this involves
a search over time it is slower than just finding any witness, although in this
particular case the extra time required is small; the same example consumed
0.6s of CPU time. As a final extension, we can ask the routine to find all
the edges in the skeleton of the common collision region. In this case there
is no way of terminating early; for the same example this process took 15.9s
of CPU time. The morale here is that a simple yes/no answer is normally
far easier to obtain!

_____

[7]A simple extension of the arguments in [CY] show that the four-dimensional S-bounds
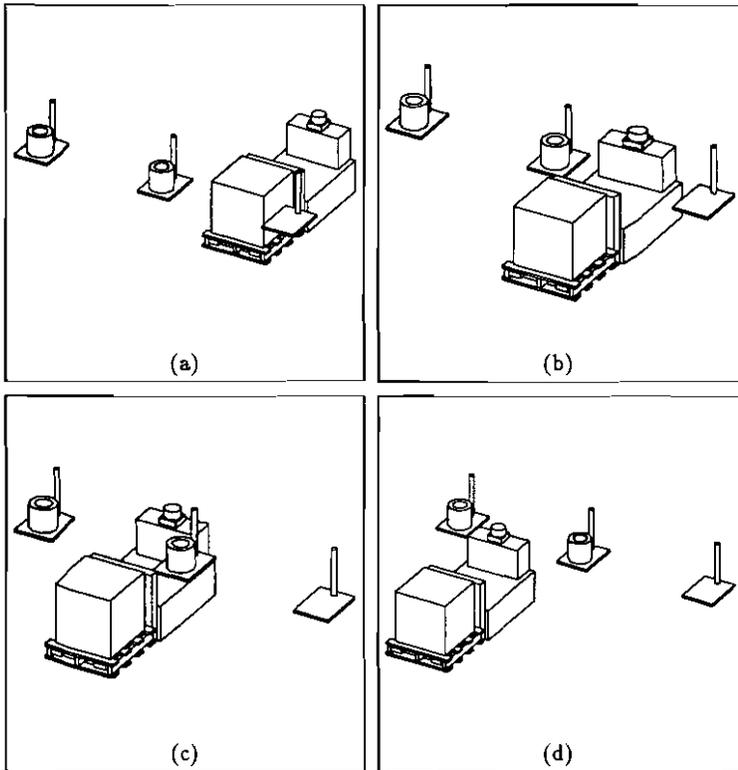must converge in a linear number of passes.

Figure 4: Vehicle Weaving

The example above only considered single motion components. In figure 4 we show the same objects as before, but now the trays have been lowered, so that the only part of the vehicle that a tray can only pass over is the centre section. The trays are moving with constant velocity, as before: the vehicle moves forward ((a)), stops((b) and (c)), and then moves forward ((d)), allowing a tray to move over itself in the process. It thus avoids collisions (but only just). The routine is able to check this; indeed, the S-bound stage is sufficient here (after two Up-Down passes), as the paths and the objects happen to be aligned with the spatial axes. To make the problem harder we can run the same test, but with all the spatial axes skewed with respect to the "natural" axes defined by the problem. Even in this case checking for a collision took 4.1s; with the S-bound stage passing on a 11 leaf tree to the divide-and-conquer stage for one of the 3 time spans, and providing 5 leaf trees for the other two time spans.

6 The Multiple Objects Problem

Up to now we have been considering the problem of finding collisions between only one pair of objects. If many objects are moving, we will wish to detect collisions between any pair of objects over a time span. A simple way of performing this is to consider each possible pair of objects separately. In many cases this is quite a sensible strategy, as we may wish to only test for collisions between certain pairs. For example, if we have a robot manipulator we can often ignore the possibility of collisions between adjacent links. However in the general case of n objects we will have $\Theta(n^2)$ possible object pairs to consider. Using extrusions it is possible to minimise any duplication of effort, using the scheme given below. We follow the order that we used in the description of the case of a single pair of objects: the theoretical basis; the divide-and-conquer mechanism; and the use of S-bounds.

6.1 Theoretical Basis

We have n moving objects, say $O_1, O_2, \ldots, O_n$, with each O_i having a location function Λ_i. To tell whether any pair collide, we need to determine whether $E_i \cap E_j \neq \emptyset$ for $i \neq j$, where $E_i = \mathbf{Ex}(\Lambda_i, O_i)$. But this will follow if the union of these $E_i \cap E_j$'s is non-null, i.e., if

$$\biguplus_i \{E_i\} \neq \emptyset$$

where $\uplus$ is a new n-ary set operation, defined by

$$\underset{i}{\uplus}\{X_i\} \equiv \bigcup_{j \neq k} X_j \cap X_k$$

(For completeness, we define $\uplus$ to return $\emptyset$ if it has less than two argument sets.) Then to consider whether there are intersections between any pair from $\{E_i\}$, we form a *single* CSG tree for each E_i, and then combine these as children of one $\uplus$ node. This operation has space and time complexity linear in the size of the extrusions.

6.2   Divide-and-Conquer

Given a composite CSG tree, with a $\uplus$ operation at the root node, the division process of §4.1 can be used with little modification. That is, starting with a tree and region of space-time, we need to be able to simplify the tree with respect to the region. Comparing the individual half-spaces at the leaves works exactly as before; to rewrite the tree, we need only additional rewrite rules for the $\uplus$ operation. These are given by the identities:

- If $X_j \equiv \emptyset$ $\uplus_i\{X_i\} \equiv \uplus_{i \neq j}\{X_i\}$

- If any pair of X_i's are equivalent to Ω $\uplus_i\{X_i\} \equiv \Omega$

The latter case yields proof of intersection.

The conquer stage is also very similar to before. We just form candidate lines in space-time by taking triples of half-spaces; split these using other half-space boundaries to form candidate edges; and consider the interior of these edges as points to check. Again, as each point is tested by considering a number of logical formulae this step is easily extended to take the $\uplus$ operator into account.

Of course there is an extra penalty for considering n objects at once, instead of two; the size of the tree considered is bigger (by a factor of $n/2$). Assuming a division strategy that reduces the problem into subproblems of bounded size then the effect of this increased size is to increase the division time only; depending on the statistical distribution of the primitives, this can be expected to increase only slightly worse than linearly [Cam84].

6.3   S-bounds

The use of S-bounds in the many-pairs case is not as straightforward. The presence of the $\uplus$ operator at the root node of the tree changes the effective topology of the expression into a graph, as it is possible to find many paths

from the root node to any leaf. However, the rewrite rules that make S-bounds efficient are defined on tree structures.

So to consider the properties of S-bounds about the $\uplus$ operator, we should rewrite the $\uplus$ as an equivalent tree. For example,

$$\uplus\{E_1, E_2, E_3\} \equiv (E_1 \cap E_2) \cup (E_2 \cap E_3) \cup (E_3 \cap E_1).$$

We note that the subtrees $E_1 \cap E_2$, etc., look like the entire tree in the case of a single pair of objects, and so if we were to concentrate our attention on one such subtree we could use the operators described there (including the special $\cap$ operation). Also, if we apply the Upward rule at the root node (a $\cup$ operation) we obtain a bound on the entire tree: this bound will be bigger than that of any of the subtrees, and so a subsequent application of the Downward rule about the root node will have no effect on the bounds of the subtrees $E_1 \cap E_2$, etc. Thus each subtree is, effectively, an island, which will receive no useful information from any of the other subtrees. This suggests a way for dealing with S-bounds about a $\uplus$ operator, *without* producing the expanded tree.

1. Let the entire tree be $\uplus\{E_i\}$. Associate with each node in each E_i an array of three-dimensional aligned boxes. To start the process, form three-dimensional S-bounds by considering just each E_i, placing the result for each node in every element, and set the four-dimensional bound for the $\uplus$ node to be Ω.

2. For every pair $1 \leq i < j \leq n$, apply the 4DSB processing for the *implicit* subtree $E_i \cap E_j$. This is done by using the scheme of §4.3.2, using the bounds in the jth array elements from E_i, and *vice versa*. When we are satisfied with the bounds formed, add the (implicit) bound of $E_i \cap E_j$ to that of the $\uplus$ operator.

The end result of this processing is a total (four-dimensional) bound at the root ($\uplus$) node, plus a set of n bounds for each node of each E_i, with the jth set of bounds at E_i corresponding to possible interactions with E_j.

It will be noted that we have considered the interaction of all $\Theta(n^2)$ pairs of objects by this process. We regard this as a necessary evil, whose effect we are trying to minimise. It is possible to produce bad-cases in which each object could, conceivably, collide with every other object, and so our routine must, in such cases, be prepared to consider all such pairs of objects. However, we believe that most real-life situations are much better behaved, and that only a few pairs of objects might collide. In such cases the S-bounds can decide, not only which pairs might collide, but also give bounds on the region of space-time in which each collision occurs, and even which parts of

each object could be involved in the collisions. Such 'normal' situations will result in most of the bounds created being $\emptyset$. Further, it is possible to disable any further consideration of collisions between certain pairs of objects (e.g., adjacent links in a robot structure) by *setting* the relevant S-bounds to $\emptyset$; this effectively prunes the relevant pair from the CSG description.

So once we have created these S-bounds we need to use them within the divide-and-conquer mechanism. Here we suggest two schemes. The first, which would work well if only a few pairs of objects are shown to be capable of colliding, simply identifies those pairs (from the S-bounds) and then tackles each pair separately (as in §4). This scheme is space-efficient, as we can process each pair when their S-bounds are considered without storing the S-bounds further. However, we are then performing the divide-and-conquer process many times. To avoid this we can use the second scheme, which is based on §6.2, but where we treat the union of the array of S-bounds stored at each *leaf* node as an outer bound for that node, and take these bounds into account during the division process[8]. This is done by ignoring any leaf node whose total S-bound does not intersect the region of interest; as the division process proceeds, the regions of interest get smaller, and so more leaf nodes are (on average) pruned out. Further, at the conquer stage we can take the S-bounds into account during the point classification stage.

The second scheme is likely to be more efficient than the first when there are a large number of possible collision regions between a large number of pairs of objects; however the organisational complexity of the scheme increases. Intermediate approaches are possible; instead of forming the exact union of the S-bounds in the array (as a list of S-bounds), we could form an approximation to the union, using $\sqcup$ instead of $\cup$. Again, the relative advantages of these approaches is heavily influenced by the geometrical domain; the intermediate approach is likely to work well if the possible collision regions for each object are localised (in space-time).

Another approach that is likely to be useful for the many-pairs case is to build up a hierarchy of approximations to the objects. For example, in [FT87] a list of approximations to the shape of objects is used, with the later approximations being finer than the earlier approximations. We may think of the early approximations as shells around the objects; their algorithm initially considers the relationships between the outer shells, and when these get 'too close' the current shell is 'broken' and the next approximation used. Thus a variable resolution is used in the models, depending on the distances

---

[8]We can, without loss of generality, consider only leaf nodes as the bounds formed are monotonic decreasing in size as we work down the CSG trees.

between different objects. To use this idea we would have to build a series
of coarse S-bounds for each approximation, and to use different levels of
approximations for different pairs of moving objects. In fact we may also
regard the S-bounds in the CSG tree as naturally forming an approxima-
tion hierarchy, although then with sub-components of objects rather than
separate objects.

7 Summary

We have introduced a formalism that allows us to model objects in motion
by subsets of space-time, and explained how the topological properties of
the objects and motions affect the extrusions formed.   Extrusions can
be used to transform the collision detection problem into an intersection
detection problem in space-time. The problem transformation is general,
but takes on a particularly easy form when the objects are described as
a set-combination of half-spaces. An implementation of the method has
been developed for the case when the objects are polyhedral and moving
with linear motions. The implementation uses a preprocessing step (based
on S-bounds) which determines interesting regions of space-time in which to
search for collisions. This step also identifies which *parts* of each object could
be involved in collisions, and hence simplifies the size of the intersection
detection problems. It should be noted that the preprocessing step is easily
extended to deal with other geometries, as we only need bounds on the sizes
of regions. It could also be used with other forms of shape descriptions, for
example, B-reps [RV82] where an S-bound is stored with every boundary
feature, although then it is more difficult to identify which subcomponents
of the objects might be involved in collisions. As a special case we could use
a three-dimensional modeller to test for collisions between two-dimensional
objects.

The output of the preprocessing step is processed by a divide-and-conquer
mechanism. This is based on splitting the original problem into a number of
simpler problems, each of which is finally tackled using a generate-and-test
routine. Of these stages only the 'generate' step is difficult to generalise
to arbitrary shapes and motions, as we used knowledge of the properties of
linear equations to produce our set of points to test.

In use the preprocessing step is seen to be efficient at selecting regions
of space-time to test, at least for objects moving with linear motions. We
conjecture that the preprocessing will also work well for general motions if
we select bounds that are the extrusions of a simple shape (such as spheres
[Cam89]) moving with the centre of mass of the objects; this will involve a

more complicated bound combination strategy as we will then, effectively, have to solve the collision detection problem for spheres.

If many objects are moving we may wish to consider many potential pairs of objects in collision. This can be tackled under the same framework by a slightly more complicated preprocessing system that identifies which object-pairs are of interest. The remainder of the processing can be performed (potentially in parallel) by a simple extension to the divide-and-conquer framework. Although for n objects there are $\Theta(n^2)$ object pairs that could collide, the advantage of our approach is that we can share much of the processing, as the extrusions for each object are the same regardless of which other object is potentially involved in a collision. This is not the case for, say, the swept volume method for collision detection, in which the *relative* motions between objects has to be used.

The main limitation of the routine described here is in terms of the shapes and motions it can consider. However the ability to deal with linear motions is useful for cartesian mechanisms and robots, vehicles, and the end-effectors of general robots under cartesian control. General rotations, such as those affected by the body of an anthropomorphic robot, do cause practical difficulties. Most of the routine is easily extended, with the real problem being performing the final null object detection tests (§4.2), which must generate a sufficient set of points to be sure of collisions. Effectively, if you double the number of different types of surfaces that have to be considered then the number of ways of generating test points goes up by a large factor, whereas the extensions to the other stages scale linearly. This effect is well-known within the geometric modelling community. A partial solution might be to adapt Canny's algorithm [Can86] as a solution to the null object detection problem, either by using his quaternion mapping to encode rotations as polynomial half-spaces, or by calling his routine in the hard cases with the vertices and surfaces within the regions given by the divide-and-conquer mechanism. Canny's implementation combines 'traditional' hand-encoded programming (to describe the configuration space obstacles) with computer algebra techniques (to find the roots of the polynomials). For a truly general solution, in terms of the coverage of surface and rotation types, we believe that we will need further advances in computer algebra and theorem proving, in order to write routines that can automatically handle the new surface types as they are added.

Acknowledgements

References

[Abb52] Edwin A. Abbott. *Flatland.* Dover, New York, 1952. Second edition, originally published 1884.

[Boy79] J. W. Boyse. Interference detection among solids and surfaces. *Communications of the ACM*, 22(1):3–9, 1979.

[CA88] Stephen Cameron and Jon Aylett. ROBMOD: A geometry engine for robotics. In *IEEE Int. Conf. Robotics and Automation*, pages 880–885, Philadelphia, April 1988.

[Cam84] S. A. Cameron. *Modelling Solids in Motion.* PhD thesis, University of Edinburgh, 1984. Available from the Department of Artificial Intelligence.

[Cam85] S. A. Cameron. A study of the clash detection problem in robotics. In *IEEE Int. Conf. Robotics and Automation*, pages 488–493, St. Louis, March 1985.

[Cam89] S. A. Cameron. Efficient intersection tests for objects defined constructively. *Int. J. Robotics Res.*, 8(1):3–25, February 1989. Similar to Oxford Programming Research Group TM-85.

[Can86] John Canny. On detecting collisions between moving polyhedra. *IEEE Pattern Analysis and Machine Intelligence*, 8(2):200–209, March 1986.

[Can88] John F. Canny. *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, 1988.

[CK86] R. K. Culley and K. G. Kempf. A collision detection algorithm based on velocity and distance bounds. In *Int. Conf. Robotics & Automation*, pages 1064–1069, San Francisco, April 1986.

[CY90] S. A. Cameron and C. K. Yap. The use of bounds in geometric processing. Accepted for publication, ACM Transactions on Graphics, 1990.

[Don87] Bruce R. Donald. A search algorithm for motion planning with six degrees of freedom. *Art. Intelligence J.*, 31(3):295–353, March 1987.

[dPBB83] A. de Pennington, M. S. Bloor, and M. A. Balila. Geometric modelling: A contribution towards intelligent robotics. In *13th. Int. Symp. Industrial Robotics*, Chicago, 1983.

[ER83] D. M. Esterling and J. Van Rosendale. An intersection algorithm for moving parts. In *Proc. NASA Symposium on Computer-Aided Geometric Modeling*, pages 119–123, Hampton (VA), April 1983. Conf. publ. 2272.

[FS89]   Kikuo Fujimura and Hanan Samet. A hierarchical strategy for path plan-
          ning among moving obstacles. *IEEE Trans. Robotics and Automation*,
          5(1):61–69, February 1989.

[FT87]   Bernard Faverjon and Pierre Tournassoud. A local based approach for
          path planning of manipulators with a high number of degrees of freedom.
          In *Int. Conf. Robotics & Automation*, pages 1152–1159, Raleigh, March
          1987.

[Gla88]   Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Comp.
          Graphics & Applications*, 8(2):60–70, 1988.

[JP88]   Deborah A. Joseph and W. Harry Plantinga. Efficient algorithms for
          polyhedron collision detection. In preparation: Department of Computer
          Science, University of Wisconsin—Madison, 1988.

[LP83]   T. Lozano-Pérez. Spatial planning—a configuration space approach.
          IEEE Transactions on Computers, C-32(2):108–120, February 1983.

[LP87]   T. Lozano-Pérez. A simple-motion planning algorithm for general robot
          manipulators. *IEEE J. Robotics & Automation*, 3(3):224–238, June 1987.

[Mey81]   Jeanine Meyer. An emulation system for programmable sensory robots.
          IBM J. Res. Dev., 25(6):955–962, November 1981.

[Mye81]   J. K. Myers. A supervisory collision-avoidance system for robot con-
          trollers. Master's thesis, Carnegie-Mellon University, 1981.

[Req77]   A. A. G. Requicha. Mathematical models of rigid solid objects. Technical
          Report PAP TM-28, University of Rochester, November 1977.

[RV82]   A. A. G. Requicha and H. B. Voelcker. Solid modeling: A historical
          summary and contemporary assessment. *IEEE Comp. Graphics & Ap-
          plications*, 2(2):9–24, March 1982.

[RV89]   J. R. Rossignac and H. B. Voelcker. Active zones in CSG for accelerat-
          ing boundary evaluation, redundancy elimination, interference detection,
          and shading algorithms. *ACM Trans. Graphics*, 8(1):51–87, January
          1989. Also as IBM Research Report RC13490, Yorktown Heights, NY,
          February 1988.

[Sam84]   Hanan Samet. The quadtree and related hierachial data structures. *ACM
          Computing Surveys*, 16(2):187–260, June 1984.

[ST85]   Hanan Samet and Markku Tamminen. Bintrees, CSG trees, and time.
          ACM Comp. Graphics, 19(3):121–130, July 1985. Presented at ACM
          SIGGRAPH Conference.

[TR80]   R. B. Tilove and A. A. G. Requicha. Closure of boolean operations on
          geometric entities. *CAD J.*, 12(5):219–220, September 1980.

[Woo86]   J. R. Woodwark. Generating wire frames from set-theoretic solid models
          by spatial subdivision. *Computer-Aided Design J.*, 18(6):307–315, 1986.

Appendix

A   Neighbourhood Computation in Four Dimensions

1. The neighbourhood is input as a Boolean function (CSG tree) over a number of linear half-space passing through a common point and all containing the line direction $\hat{l}$. By sorting the normals to the half-spaces we can indentify any multiple references to a half-space or its complement, and so express the Boolean function as a function of a unique set of normals $\{\hat{p}_i\}$ for $1 \leq i \leq n$. In turn by forming an orthonormal basis $\{\hat{l}, \hat{e}_1, \hat{e}_2, \hat{e}_3\}$ and expressing each $\hat{p}_i$ as $\langle 0, p_i \rangle$ we convert the neighbourhood into the equivalent three-dimensional case of number of planes passing through the origin.

2. If $n \leq 2$, or if $n = 3$ and the $\{p_i\}$ are linearly independent then the logic-based approach suffices [Cam89]. (Linear dependence is easily checked by generating the basis vectors $\hat{e}_i$ from the $\hat{p}_i$ using the Gramm-Schmidt process.)

3. If $n = 3$ but the $\{p_i\}$ lie in a plane, the following is useful. Consider the Boolean function and count for how many of the 8 possible inputs it can return **true**. If the answer is 0 or greater than 2 then we can be sure whether the neighbourhood is empty or not, as there are exactly 2 spatially redundant cases.
   (In practise steps 1–3 take care of the vast majority of cases.)

4. Otherwise, if all the $\{p_i\}$ lie in a plane consider a new set of $2n$ test points of the form $\{\pm p_i \times n\}$, where $\times$ is the vector product operator and n is the normal to the common plane. Each test point must be amenable to the logic-based approach.

5. Similarly, if the $\{p_i\}$ span three dimensions consider test points of the form $\{\pm p_i \times p_j\}$. This is equivalent to testing a general vertex in three-dimensions by crawling along all possible edges leading from that vertex, and testing those recursively.

B   Special Form of the ⊓ Operator

The routine to he described takes two S-bounds, each consisting of a spatial (rectangular) bound moving with coustant velocity, and computes two new spatial bounds that *tightly* enclose the intersection, together with a new

time-span $[t_l, t_h]$ over which they are valid. We may obtain a null time-span, which indicates that the space-time bound is null, and thus that the objects cannot collide (in this time-span). The algorithm proceeds as follows:

1. Compute the new time-span, $[t_l, t_h]$, by considering the intersection of the spatial bounds.

2. By considering each spatial dimension separately, compute the new spatial bounds.

Note that if the objects have the same velocity, then the temporal bound is unaffected, and the change in spatial bound is equivalent to that for the three-dimensional S-bound system.

Computing $[t_l, t_h]$ Let q be one of the spatial parameters (x, y or z). Then if we ignore the other spatial parameters we are given four relationships between q and t, of the form

$$ut + a \leq q \leq ut + \alpha \qquad\qquad vt + b \leq q \leq vt + \beta$$

(a, α, b and β are obtained directly from the 3DSBs, and u and v are the velocity components.) Solving these inequalities for t gives $b - \alpha \leq (u-v)t \leq \beta - a$. $[t_l, t_h]$ is formed by taking the intersection of the three intervals formed in this way, together with $[t_L, t_H]$. (A null time interval causes a null set to be returned, signifying a provably null region.)

Computing the Spatial Bounds For each spatial component q, we effectively compute bounds on q at each of t_l and t_h, and then "push" the four spatial bounds to touch these bounds. The bound at t_h is given by $q_h \leq q \leq Q_h$, where $q_h = \max(ut_h + a, vt_h + b)$, $Q_h = \min(ut_h + \alpha, vt_h + \beta)$, and we can obtain similar expressions for the bounds at t_l, q_l and Q_l. (Note that, by our choice of t_l and t_h, $q_h \leq Q_h$ and $q_l \leq Q_l$.) Then we need to choose values for the new spatial bounds, $[a', \alpha']$ and $[b', \beta']$, so that the relevant space-time bounds ($ut + a' \leq q \leq ut + \alpha'$, etc.) contain the intersection region. This is satisfied by setting $a' = \min(q_l - ut_l, q_h - ut_h)$, which simplifies to

$$
\begin{aligned}
a' &= \max(a, b + \min((v - u)t_l, (v - u)t_h)) \\
\alpha' &= \min(\alpha, \beta + \max((v - u)t_l, (v - u)t_h))
\end{aligned}
$$

with similar expressions for b' and β'. (To derive these forms, apply the affine transformations $q \rightarrow q - ut$ and $q \rightarrow q - vt$ in turn. Note that we are guaranteed to have $a' \leq \alpha'$, etc.)
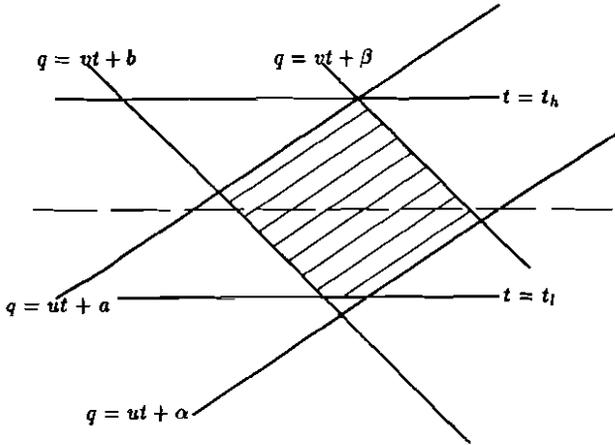
Figure 5: Example of a projection in q–t space

An example is given in figure 5, in which the dashed region shows the intersection of the six bounds, but for which a reduction in the size of the bounds is not possible (along this spatial dimension). If one of the temporal bounds were to lie along the dashed line instead, a reduction in size would be possible.

Optimal Fitting of the Final Region   The final region is the intersection of the two 4DSBs, together with a time bound, $[t_l, t_h]$. We need to find a single rectangular bound around this region, to be passed to the divide-and-conquer routine. However, we are at liberty to measure the velocities with respect to any frame we choose when selecting this frame. This is equivalent to applying an affine transform to the space-time diagrams, or fitting an optimal parallelogram region around the projection of each parameter q. In fact, if we choose to measure with respect to a frame moving with velocity w in the direction above, and noting that the intersections of the left-most and right-most bounds in q cannot be redundant (as otherwise we could choose better bounds), then we first see that choosing w outside the range between u and v cannot give a optimal fit. So consider $w = \lambda u + (1 - \lambda)v$ for $\lambda \in [0, 1]$. Then we can show that the sides of the parallelogram are Δc

apart (measured in the q direction), where

$$\Delta c = \lambda(\alpha - a) + (1 - \lambda)(\beta - b)$$

and so we are best choosing between $w = u$ or $w = v$ (unless either is optimal, in which case so is any such w). Notice that this is not necessarily the same as choosing to regard one of the objects as fixed: we decide which object to "fix" in each spatial dimension separately.

.