To my parents
who provided me
such a fine start in life

Diploma Thesis

Exact and Fast Collision Detection

Gabriel Zachmann

1994

Contents

Chapter 1

Introduction

1.1 Collision Detection — What For?

Collision detection has many different applications; for example, in *physically based simulation*, where moving objects are simulated. In order to determine their behavior over time, the most basic information needed is the time and position of collision together with the exact point of collision. Only if this information is known exactly, the collision response can determine how objects will react, according to their mass, mass distribution, velocities, etc.

Animation is one of many applications of physically based simulation. An animation system does not really need absolutely correct physical simulation; usually it suffices that the behavior and paths of objects look realistic. An animator specifies weight, initial speed, and maybe an inertia tensor, too, and has the simulation module compute the paths of the objects. Then he compares those with the paths he had in mind and verifies that they look realistic. If so, he's done; otherwise he tweaks some parameters of the objects until the output matches with his intention.

Another area is robotics. A common application is *path planning* [ELP87]: given the geometry of an object and the start and end point of the motion of it, the task is to find a path among several obstacles so that the object will never hit any obstacle.

For *tele-operation*, the goal is usually to avoid any collision between the tool (e.g., a robot arm), which is remotely operated, and any fragile obstacle [CAS92].

NC milling and *CAD* also utilize collision detection in order to make sure that the path of the milling tool does not cut off any parts with the back of the tool [HL92].

In *virtual reality*, collision detection can be used to facilitate intuitive interaction [ADG+94], natural manipulation of the environment, any kind of physically based simulation, and modeling. In general, collision detection with appropriate collision response can make a virtual reality application look more believable [Hubbard93] (which is supposedly why it is called virtual "reality"), because collision detection is usually the first step towards objects behaving more "real".

1.2 Classification

1.2.1 Object Representations

The internal representation of graphical objects has great impact on the choice of algorithms, not only for collision detection, but also for rendering, modeling, and many other parts of an interactive graphical system.

The last 15 years, several different approaches to object representation have emerged. A very basic classification is *boundary based* vs. *volume based*.

Boundary based object representations are the classical b-rep (see also Section 4), free form surfaces, "augmented" octrees [CCV85, NAB86, FK85], and hierarchical b-reps (see Section 2.1).

Among *volume based* object representations are the well-known octree [YKFT84, TS84, FA85, NAB86, Dyer82, TKM84], BSP [PY90, TN87, NAT90, Torres90, Vanecek Jr.91], and CSG. Less known representations are sphere splines [MT], which approximate an object by moving a sphere along some spline curve while varying its radius; ray-reps [MMZ94], which represent an object by a collection of parallel line segments of varying length and position; and the H-P model, which approximates an object by slices of a sphere [CM87].

Other representations are *primitive instancing*, *constructive solid geometry* (a generalization of primitive instancing), and *sweep representations*.

Discussion

All of these object representations have been devised to suit special needs. BSPs are suitable for hidden-surface removal without z-buffer hardware and have found their way into solid modellers. Free form surfaces are good for modeling curved surfaces with higher continuity. Octrees are also used in solid modellers and for representing volume data.

Octrees and BSP trees are well suited for intersection computation of polyhedral objects; however, we are not really interested in the complete intersection of two polyhedra, which is another polyhedron (the *construction problem*), but only in solving the *decision problem* whether there is an intersection of two polyhedra or not.

While BSPs represent an object exactly by intersections of half spaces, octrees approximate an object by a hierarchical decomposition into cubes. Thus, octrees cannot provide for exact collision detection algorithms (unless we use augmented octrees).

Neither BSPs nor octrees seem to be suitable for objects which change their geometry, because in this case their BSP- or octree-representation would have to be re-computed. Also, octrees have to be recomputed when the object moves, which is particularly expensive, because the motion includes rotation (octrees are based on axis aligned cubes). In any case, a b-rep representation seems to be needed in addition to a BSP or octree representation, at least for the renderer. Thus, in an integrated system, two representations would have to be maintained, which is always prone to inconsistencies, increases memory requirements, and might destroy any efficiency gain.

Furthermore, in order to achieve a fairly reasonable accuracy with octree representations, quite a bit of memory has to be spent.

The advantage of b-reps is that they can easily hold topological information about the geometry, like adjacency and incidence of features (i.e., vertices, edges, polygons). Octrees are better suited for computing mechanical properties like mass, volume, inertia tensors, etc. [LR82, TKM84]. However, if the geometry does not change, these properties can be pre-computed at load time, and then the octree can be discarded.

Sphere splines and sphere coverings (the discrete variant of sphere splines [OB79]) are also only approximate representations; thus they cannot provide for exact collision detection algorithms.

The considerations above have led to the decision to use solely b-reps for polyhedra, and to build any additional data structures, which might be needed for speed-up, on top of these b-reps.

1.2.2 Types of Collision Detection

In a typical application using any kind of collision detection, there are two major parts within the *collision handling* module: the *collision detection* and the *collision response*. The collision detection part determines whether simulated objects would penetrate each other, while the collision response part uses the output of the former part to prevent this to happen. Although both parts pose interesting problems, we will focus only on the collision detection part (except in Section 8). For further reading on the collision response part see [MW88, BJ91, Hahn88].

Collision detection algorithms can be classified by several criteria:

- *approximate* vs. *exact*; approximate collision detection is usually biased, i.e., the algorithm tends to favor one answer over the other (see Section 3.4.2 and 3.5). The bias is caused by using some sort of geometry simplification or a probabilistic algorithm.
- *time as a fourth dimension* vs. purely three-dimensional, *timeless* geometries; approaches which take time into account are [Hubbard93, Canny86].

  The time dimension can be used to compute the exact time of a collision, or it can be used to exploit time coherency in order to speed up the collision detection procedure.

  Timeless approaches consider all objects only at a certain time, but they do keep in mind that objects probably move — unlike approaches in computational geometry. If timeless approaches have to provide the time of collision more accurately, they will resort to some kind of back-tracking method.
- speed-up by object *hierarchies* [YW93] (above object level or on intra-object level), by space subdivision (see Section 5), or by plane sweep [MS85a, AS90].
- *restriction of the domain*; mostly, the class of polyhedra is restricted to convex ones. Other possible restrictions could be closed objects (see Section 3.5) or polyhedra consisting of convex polygons (see Section 3.3).
- *flexible* vs. *rigid* objects; the issue of collision detection between flexible (also called "soft") objects [Gascuel93, SWF+93], which change their geometry with time, complicates the problem significantly, because either no pre-computation can be used at all, or the pre-computation has to be updated with every change of geometry. Self-intersections might have to be checked for, too [MW88]. Some algorithms are based on parametric object representations (e.g., free-form surfaces), while others still use b-reps.

- *on-line* vs. *off-line*; many applications can do without on-line collision detection, because the application is not driven by real-time input like in VR environments, for example, path planning in robotics or physically based simulation for animation.
- *incremental* vs. *"from-scratch"*; incremental methods try to exploit results of an earlier collision query (see Section 3.4.2 for an example). This is a form of exploiting time coherency.

For a short discussion on different types of collisions see Section 3.1.

With robotics, collision detection usually does not have to be real-time or exact; usually path planning can be done off-line, and it suffices if the path makes sure that there won't be any collision. However, if there are moving obstacles whose motions are not known in advance, path planning will become more like on-line collision avoidance in a not fully predetermined environment.

Also, animation, NC milling, and CAD usually do not necessarily need real-time collision detection. It is desirable, however, to have collision detection as fast as possible: then, the path of the milling tool can be adjusted interactively during the planning phase; with animation systems, the properties of objects (mass, initial speed, etc.) can be modified on-line by the animator until the path computed by the simulation module matches what the animator originally had in mind.

Collision avoidance with tele-operation does need real-time collision detection. However, usually it does not need exact detection: since there should always be some safety distance between tool and obstacles, there is no point in doing exact collision detection. So, the avoidance system may as well utilize approximate object representations, which help a lot in speeding up the task [CAS92].

In virtual reality, the requirements are most severe. Under all circumstances, the collision detection must be real-time in order to attain the effect of immersion. For physically based simulation within a VR environment, it is also highly desirable to have exact and accurate collision detection, because there won't be a second chance to tweak if the output of the simulation module is not satisfactory. Although interaction in virtual environments (e.g., pressing 3D buttons) might not really need the exact point of collision, a detection too inaccurate (e.g., only bounding box tests) disturbs the impression of realism.

Common difficulties with collision detection

There are several difficulties that commonly arise when the ultimate goal is real-time exact collision detection:

- *pairwise* tests, on the object level as well as on the edge-/face-level; a naive algorithm has to test all possible pairs of edges and faces, and also all possible pairs of objects [Hubbard93].
- *discrete time*; this makes it hard to compute the exact time of collision.

  Dynamic graphical systems display all objects at certain time intervals, usually as soon as the application is done with all the computation, like gathering input data, simulating the environment, moving objects, etc. If the collision detection module "sees" the environment only at these time steps without any further information about the future, then it can only check whether there is a collision or not.

  If speeds (translational and rotational) and maybe acceleration are provided, too, then the module can also compute the exact time of collision (or an approximation thereof), either by computing the next collision in the future, or by back-tracking and recursive interval bisection (of the time interval).

Recursive interval bisection is in fact a sort of a root-finding algorithm, which could fail if objects moved too fast. However, this is usually not a problem, since dynamic systems always try to make time steps as small as possible.

- *concave* polyhedra, or, even worse, polyhedra which do not consist of convex polygons and which are not closed.

  There are many possible ways to tackle the collision detection problem with convex polyhedra; for non-convex polyhedra, very few algorithms seem to be known. For closed polyhedra, we can still resort to algorithms for convex polyhedra only, by partitioning them into convex pieces; if they are not closed, there is little we can do.

Classes of polyhedra

Here, we only consider the class of polyhedra which can be represented by b-reps, i.e., we do not consider curved surfaces (algebraic or parametric).

- a *collection of polygons*; in this class, we do not require anything but plane polygons (objects don't even have to be closed).

  If we allowed polygons whose vertices do not necessarily lie in a plane, the class would be even larger. Exact collision detection for this class would be very hard; but then, the definition of a polygon with more than four non-coplanar vertices is unclear, too.

- *closed polyhedra*; in this case, an "inside" and an "outside" can be defined and exploited.

- polyhedra consisting of only *convex polygons*; these polyhedra may still be non-convex.

- *star-shaped* polyhedra [Toussaint88]; this class does not seem to yield any advantage with respect to collision detection.

- *convex* polyhedra; this class is probably the smallest reasonable one. It seems to provide the greatest advantage for incremental algorithms [LC92, LC91, LM91].

General methods how to speed up collision detection

There are several features that can be exploited to speed up collision detection, some of them at the pairwise collision detection level, some of them at the global level, which is concerned with multiple moving objects:

- All kinds of *bounding volumes*, for objects (see Section 5) as well as on the polygonal level (see Section 3). The most common bounding volumes are axis-aligned boxes, others are bounding spheres and non-axis aligned bounding boxes (see Section 5.2).

- *Space coherency* uses the fact that usually large regions of the space are occupied by only one object or none at all. This is the basis for speed-up on the global, multiple-object level by space indexing methods.

- *Time coherency* exploits the fact that moving objects usually move on a (conceptually) continuous path. Also, every interactive system will try to maximize the frame-rate, so that objects move rather slowly compared to the frame-rate. If the path is even $\mathcal{C}^2$ continuous and the acceleration can be estimated in advance, this can be exploited by the use of space-time bounding volumes [Hubbard93].
- *Restrictions of the input class* of polyhedra (see paragraph above).
- *Pre-processing*; by augmenting geometry data with additional data structures, faster algorithms are made possible. The major drawback with this approach is that any algorithm based on pre-processing cannot be used whenever geometry changes often during run-time.
- Introducing *bias*, which is usually caused by the use of a probabilistic algorithm (see Section 3.4.2, Section 3.5). It could also be created by approximate representations.
- *Approximate representations* of the object geometry. Thus, collision detection algorithms have to deal only with very special "geometries", which make them very fast. Depending on the approximation, the algorithm is usually biased either towards the "no collision" side or towards the "collision" side.

  The approximate representation might be explicit. Stopping after a certain number of pre-check stages implies an implicit approximation of geometry (see Sections 3.2.1, 3.7.1). Using only parts of the accurate object representation is another kind of implicit approximate representation (see Section 3.4.2).
- *Parallelization*; there are several possibilities: parallelization on the pairwise edge-face intersection level (fine-grain); parallelization on the global, multiple collisions level (coarse-grain); concurrent run of two dual algorithms, one being generally fast if no collision occurs, the other being fast if collision does occur; and concurrent run of the application and the collision detection module.
- *Adaptive* collision detection; if the overall frame rate decreases below a "tolerable" threshold, then the collision detection module could switch over to non-exact algorithms.
- *Time-stamps* help to avoid looping over all faces or objects in order to mark some data invalid.

Requirements of collision detection

In general, it is highly desirable that collision detection have the following qualities. It should be:

- fast (if possible, real-time),
- suitable for a class of polyhedra as large as possible,
- exact (i.e., the module reports a collision if and only if there is an intersection of surfaces),
- able to report a witness, i.e., an edge and/or polygon, where the two objects collide (if possible, report all collision points),
- able to handle many moving objects.

Definition

For a definition of the term "*collision*" see Section 3.1.

1.3 Outline of the Thesis

This thesis' goal is to investigate collision detection in virtual reality systems, which impose real-time requirements. Precise and high-speed collision detection algorithms are developed and elaborated. Several aspects of collision detection are considered, namely object topologies, dynamic environments, and parallelization. A module for collision detection has been implemented and integrated in both a new renderer and an existing input/output-toolkit for virtual environments. All algorithms have been built on top of the new data structure specified within the framework of the Fraunhofer demonstration center for virtual reality. An application has been developed using the new collision detection module.

The remainder of this thesis is structured as follows:

Section 2 reviews previous work done in this area, but with widely varying backgrounds (computational geometry, CAD, robotics, modeling, virtual reality).

Section 3 discusses and develops various algorithms which detect a collision of a pair of objects. Algorithms are developed for arbitrary and convex topologies. A new algorithm for arbitrary objects is developed which improves previous algorithms by pre-processing. New probabilistic algorithms are developed for convex and closed objects; these are further enhanced to exploit temporal coherency. All of the algorithms just mentioned have been implemented. Two more algorithms are elaborated, which have not been implemented, yet.

Section 4 introduces and discusses a data structure which allows objects to be classified with respect to their topology and geometry. This data structure has been incorporated into the collision detection module as well as a full object classification.

Section 5 develops and discusses data structures and algorithms to solve the n^2-problem on the object level. All of them exploit spatial coherency. Two data structures have been implemented and thoroughly tested, namely octrees and grids.

Section 6 describes how to parallelize all algorithms presented so far. Four different parallelization schemes have been implemented and tested.

Section 7 gives an overview of the collision detection module which has been implemented, documents the interface, and gives some implementation details; in particular, the time-stamp technique is described, which has been made heavy use of in the module.

Section 8 presents the application using collision detection and discusses some problems with modeling in a virtual environment. This application benefits greatly from the data structure developed in Section 4.

Section 10 draws conclusions and elaborates on directions for further research.

All implementation was done in the framework of the new version of IGD's *Virtual Design*, internally called the *Y* system.

1.4 Preliminaries

A few remarks on notation, which will be used throughout this document:

$\boldsymbol{v}$	vector or point, almost always $\boldsymbol{v} \in \mathbb{R}^3$
P	polyhedron, or just a graphical object
V_P, E_P, F_P	the set of vertices, edges, and polygons of P, resp.
iff	= if and only if

A few algorithms will be presented also in some pseudo-code notation, which uses the following conventions:

$\forall \ldots :$	*a loop over all elements of a set*
`i = x...y :`	*a loop, too*
condition $\longrightarrow$ ...	*if condition is true, then ...*
$\uparrow$ *expression*	*return expression*

Chapter 2

Previous Work

Collision detection seems to have attracted much attention for the past 15 years. The first researchers came from the area of robotics and computational geometry. Despite its comparatively long history, real-time exact collision detection has not been tackled except for the past one or two years.

Later on, physically based modeling and animation had a special need for exact collision detection. Also, modellers and CAD systems in general usually provide a means of calculating the intersection of objects.

2.1 Computational Geometry

When computational geometry directed its attention to the problem, the goal at first was to *construct the intersection* of two polyhedra. Only later on, researchers realized that the *detection problem* is interesting by itself and can be solved in fact more efficiently than the construction problem.

Being interested in theoretical results, the main goal of research in this area has been to find algorithms with optimal asymptotical worst-case complexity.

All results apply only to the strictly static case, i.e., all geometrical features of an object are given in world coordinates, and its orientation/location may be assumed to be in some special position without loss of generality.

Furthermore, almost all algorithms consider only convex polyhedra.

Construction algorithms

The first to present an algorithm which has an asymptotical complexity below the trivial $O(n^2)$ were [MP78a].

Like many linear programming algorithms, the algorithm consists of two phases: the first one searches for a point in the intersection of the two polyhedra, the second phase then constructs the actual intersection (if any) by taking the dual of the two polyhedra, forming the union of these duals, and finally computing the dual of the result again, which yields the intersection.

The really involved part is the first phase. First, the border of each polyhedron is projected onto the xy-plane (see Figure 2.1); by walking along these meshes thus generated and exploiting the fact that the polyhedra are assumed to be convex, the intersection can be constructed.

The overall complexity is $O(n \log n)$ (including transformation in this case).

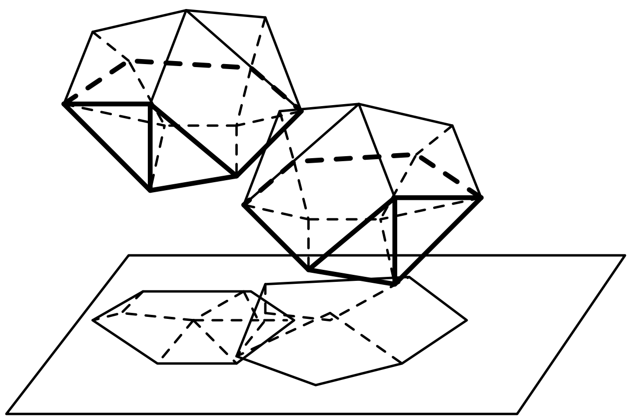


Figure 2.1. The projection of polyhedra onto a plane can be used to solve the intersection construction problem.

Another approach uses the notion of distance of polyhedra (see Section 3.1) [DK85]. In order to compute this distance efficiently, a hierarchical b-rep representation of convex polyhedra is devised (see Figure 2.2). It is defined as follows: Let P be a polyhedron with vertices V; a sequence $P_1, \ldots, P_h$ is a *hierarchical representation* of P iff

1. P_1 is a tetrahedron and $P_h = P$,
2. $P_i \subset P_{i+1}$,
3. $V_{P_i} \subset V_{P_i+1}$.

The last condition ensures that the transformation of the hierarchical representation of a polyhedron takes as much time as transforming the polyhedron itself.

A hierarchical representation can be obtained by the following simple algorithm: start with the polyhedron, find an independent set of vertices, remove these, and build the convex hull of the remaining ones. The new polyhedron is the next "lower" element in the sequence. In two and three dimensions, there is always such a hierarchy (not so in higher dimensions).

The following lemma is the heart of the iterative algorithm. Given two hierarchical representations $P_1, \ldots, P_h$ and $Q_1, \ldots, Q_k$; let H be a plane supporting P_i, then either

1. H supports P_{i+1}, or
2. $P_i \cap H^-$ is a pyramid whose apex has degree at most d,

where d is the degree of the hierarchical representation (it is somewhat similar to the degree of a graph).

The basic idea of the algorithm for finding the distance is to find the separating slab. Given two hierarchical representations, we start with a slab separating the two tetrahedra at the bottom of the hierarchies, resp., (this can be found trivially in $O(1)$). Then we proceed to the next polyhedra in the hierarchy; we check if the slab still separates these two. Otherwise we can quickly compute a new separating slab (by the preceeding lemma).

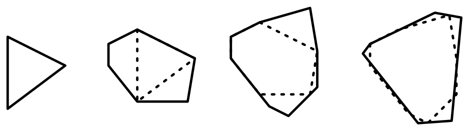


Figure 2.2. A hierarchical representation of b-reps can be utilized to calculate the distance of polyhedra efficiently.

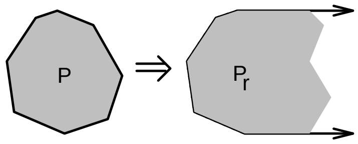


Figure 2.3. Decomposition of a convex polygon into two monotone polygonal sectors (here only the left one is shown).

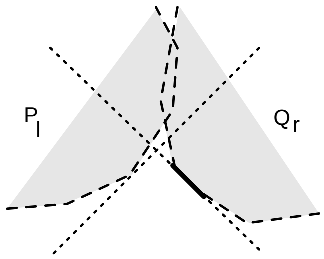


Figure 2.4. One (possible) phase of the intersection detection algorithm for two convex polygons. We can discard the lower half of P_r.

The only reference I found in the area of computational geometry which can handle a broader input class is [MS85b]. They give an algorithm which does construct the intersection of two polyhedra one of which can be non-convex. They use also the hierarchical representation of polyhedra and achieve the same upper bound.

Detection algorithms

There are also a few works on the detection problem itself [DK83, CD87, Reichling88]. The best upper bound for the detection problem (to my knowledge) is given by [DK83]. The algorithm has worst-case complexity of $O(\log^2 n)$, $n = |V|$.

However, the algorithm seems to be very involved, and no implementation is known to me. Also, it assumes the polyhedra to be pre-processed in a certain way (see below) which does not lend itself directly to moving objects.

A basic step of the algorithm is the intersection of two convex polygons in 2-space. This can be detected with $O(\log n)$ time.

The algorithm for this problem decomposes the two polygons P and Q into two monotone polygonal sectors P_l, P_r and Q_l, Q_r, resp. (see Figure 2.3), such that $P = P_l \cap P_r$, and $Q = Q_l \cap Q_r$, resp. Then, P and Q intersect if and only if $P_l \cap Q_r \neq \emptyset$ and $P_r \cap Q_l \neq \emptyset$.

The remaining problem is to find out whether or not two monotone polygonal sectors intersect. This is done by a divide-&-conquer algorithm. We start with an edge of P and an edge of Q in the middle of the sectors, resp. If the two edges intersect, we're finished. If they don't, we can determine one half of the set of edges of one of the two polygons which we don't have to consider any further for intersection (see Figure 2.4).

The next tool of the algorithm is a pre-processing which decomposes polyhedra into *drums* (see Figure 2.5). They do not necessarily have to be parallel to any of the coordinate planes. Given a drum D and a polygon P, $O(\log n)$ operations suffice to detect whether or not they intersect. The algorithm for this (sub-)task is quite the same as the one above for two polygons. Conceptually, the intersection of the drum D and the supporting plane of P is needed (which yields a polygon Q in the same plane as P). However, in order to achieve the complexity mentioned

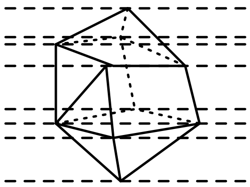


Figure 2.5. Decomposition of a convex polyhedron into drums.

above, the vertices of Q are not stored explicitly (that would introduce $O(n)$ time). Instead, whenever the algorithm would need a vertex of Q, this is computed on-the-fly. Since the edges of D can be stored in a circular enumeration, this is indeed possible in time $O(1)$.

The next sub-task is the intersection of two drums. This is done by a similar algorithm as in the two-dimensional case for two polygons. Drums are decomposed into two (infinite) half-drums. This can be done in time $O(\log n)$.

Finally, the polyhedron intersection detection is reduced to finding two drums which intersect. By an algorithm very similar to the one given for two polygons, we can detect an intersection with time $O(\log n^2)$, $n = |E_P| + |E_Q|$. We check the two middle drums of each polyhedron for intersection. If they do intersect, we're finished. If they don't, the sub-algorithm gives a separating plane. With the help of this plane and the two drums, we can determine which half of the drums of one of the polyhedra we can safely discard.

A related issue is finding intersections between two sets of spheres. This is treated by [HSS83], who give a plane sweep algorithm for an efficient solution.

2.2 Configuration Space

In the field of robotics, a completely different approach has been pursued: collisions are detected in *configuration space* (see [ELP87], for example). The approach seems to be well suited for path-planning where only one object moves (e.g., a robot manipulator).

The basic idea is to represent the moving object M as a point in a so-called configuration space. In configuration space, all obstacles O_i are usually still represented as (other) polyhedra, however, depending on the accuracy needed, they might even become curved surfaces. Configuration space will be constructed such that it has the following property: a collision of the M with an obstacle O_i in "real" space is equivalent to the situation where the moving "point" (representing M) is inside an obstacle O'_i in configuration space.

Let's assume, for the time being, that "real" space is 2-dimensional. If M is only allowed to translate, then the configuration space can be obtained by calculating the Minkowsky sum of the obstacles and the M (see Figure 2.6). This configuration space is still 2-dimensional.

If M is also allowed to rotate, then for every possible rotation of M there will be a 2-dimensional slice of the 5-dimensional configuration space. This slice in turn is the Minkowsky sum of all obstacles with M in that particular orientation.

Moving obstacles can be incorporated in this method by computing a configuration space-time, which is usually a sequence of configuration spaces at certain time steps, where the position and orientation of all obstacles is known.

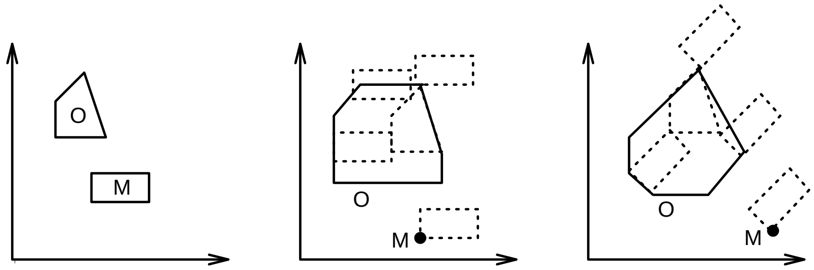


Figure 2.6. A moving object M and an obstacle O (left), and two configuration space slices corresponding to two different orientations of M.

Using this representation, path planning is solved by finding a curve (piecewise linear or smooth) through configuration space, that does not intersect any of the configuration obstacles. Additional constraints can be applied, too, like maximizing the minimum distance to any obstacle – thus path planning is an optimization problem.

2.3 Non B-Rep Approaches

As stated earlier [GASF94], the representation of objects has big impact on collision detection algorithms. Other representations, e.g., octree, BSP, CSG, etc., need quite different approaches.

Octree. Using plain octrees, any boolean operation becomes trivial [NAB86].

In particular, an intersection algorithm can be described as follows: start with the bounding box of the two objects. Given two octree(-subtrees) A,B for intersection, check if the root boxes overlap. If they don't, the intersection is empty. If they do, check for three cases:

1. the root of A is black: the intersection of A and B is B clipped by the root box of A;
2. the root of A is white: the intersection is a white box $\mathrm{bbox}(A) \cap \mathrm{bbox}(B)$;
3. both the root of A and B are black: intersect each son of A with each son of B.

More elaborate octree schemes can represent b-reps exactly [CCV85]. The basic intersection algorithm is the same; the difference is the treatment of some leaf types: if they are black or white, they will be treated the same way as with ordinary octrees. However, if both of them are non-homogeneous, the result has to be computed by a simple intersection algorithm which is capable of intersecting two planes, or a plane and an edge, etc.

BSPs. BSPs have been used for CAD systems as well, where boolean set operations have to be performed quite often [NAT90, TN87].

An intersection computation of two BSP trees is done by merging the two trees. Graphically, this operation is equivalent to overlaying the two trees (see Figure 2.7).

Given two BSP trees Q and B for intersection, the algorithm does a traversal of the BSP tree of object A. At every node of A, the tree of B is partitioned by the plane at that node. The two halves are then "inserted" at both children.

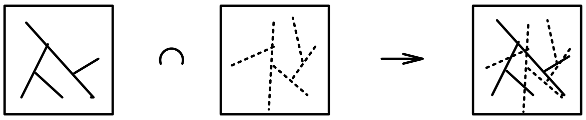


Figure 2.7. Graphical demonstration of the intersection of two BSP trees.

2.4 Approximate Algorithms

For collision avoidance systems, an approximate collision detection is quite appropriate. [CAS92] use an octree to represent all the objects of a tele-operation system. In order to be able to update this octree quickly, objects are approximated by primitives; they use "cylspheres", cylinders with spheres on each end.

The octree is constructed and maintained dynamically: nodes of the tree are split only when necessary, i.e., if it contains too many objects; and nodes are merged into one larger node, if the total number of objects inside them drops below a certain threshold.

A rather appealing approach to solve the collision avoidance problem quickly is done by [YK86] through table look-up. If there is only one moving robot manipulator with not too many joints, this approach seems to be fast.

The idea is to consider the position/orientation of the whole robot arm as a single point in configuration space (i.e., every joint adds one dimension). Conceptually, a collision detection is done for every possible joint configuration at initialization time of the collision table. Later on, a collision detection query would just take the current joint configuration as index into this table and thus get the result immediately.

Since the table is, of course, discrete, the robot has to be enlarged by the "resolution" of the table, in order to have still reliable collision avoidance. Furthermore, this works only for a static environment.

The hierarchical, approximate representation developed by [Hubbard93] is somewhat similar to the box-tree hierarchy (see Section 3.6, and also Section 2.10). He uses spheres instead of boxes, and the tree is constructed from bottom to top. First an octree is constructed for an object, for which a sphere tree is to be built. The leaves of this octree are enclosed in spheres which will become the leaves of the sphere tree. These leaves are then enclosed in larger spheres which contain their children completely.

2.5 Distance Based Algorithms

A few algorithms use the notion of *distance* between polyhedra (see Section 3.1). Clearly, two polyhedra do not intersect each other if their distance is greater than zero. All of the algorithms presented in the literature can handle only convex polyhedra.

[GJK88] present an algorithm to compute the distance between convex polyhedra with approximately linear complexity. The algorithm is also suitable for computing the distance of the spherical extension of convex objects.

The basic idea is to compute a sequence of polyhedra for every given polyhedron. This sequence is chosen such that the distance of a point to the original polyhedron

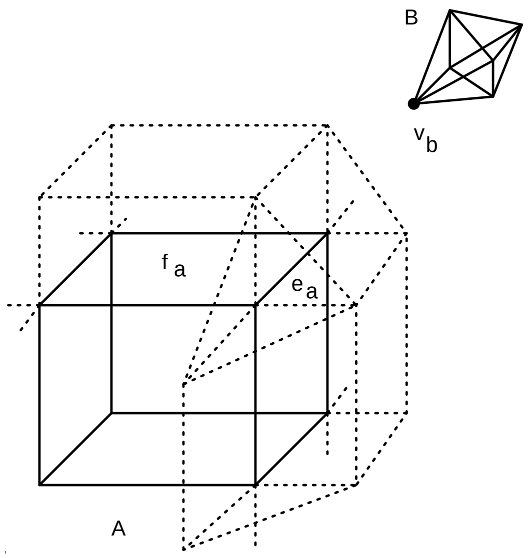


Figure 2.8. An example of the Voronoi pre-processing for a convex polyhedron. v_b is not in the Voronoi region of f_a, so (v_b, f_a) are not *realizing* features. In fact, the algorithm will replace f_a by e_a.

can be obtained by an iteration over all polyhedra in its associated sequence. With each step the distance of the point to the particular polyhedron is calculated using results from the previous step. This sequence of distances will converge to the distance of the point from the original polyhedron.

[LC91, LM91] elaborate further on this idea by building an incremental algorithm. It does not need an associated sequence of polyhedra. The idea is to maintain a pair of closest features (vertex, edge, or polygon) with every pair of objects. Since, in general, objects move slowly compared to the frame-rate, these features are probably still the closest ones the next frame, or, if not, the closest features are in the neighborhood of the old ones.

The polyhedra have to be pre-processed in order to be able to maintain the closest features quickly. To this end, the *Voronoi* region to each feature is constructed. The *Voronoi* region of a feature is the set of points which are closest to this feature [PS85] (see Figure 2.8).

Let (a, b) be a pair of features of polyhedra A and B, resp., and (p_a, p_b) two points, with $p_a \in a$, $p_b \in b$. Then (a, b) together with (p_a, p_b) *realize* the distance of A and B, $\Leftrightarrow$ p_a is in the Voronoi region of b and p_b is in the Voronoi region of a.

If two features (a, b) are given, we can easily check whether or not they do realize the distance of A,B. If they don't, we trivially know another feature (either of A or B, whichever feature fails the test) which is closer to the other one. This is a by-product of the test. This "closer" feature is incident to the one which was discarded. We iterate this until we find a feature which is closest. Because the object is convex, this will be the global optimum (not just a local one).

2.6 Object Hierarchies

Bounding volume hierarchy. If objects actually consist of several parts which are organized in a tree, we can try to maintain the corresponding bounding box

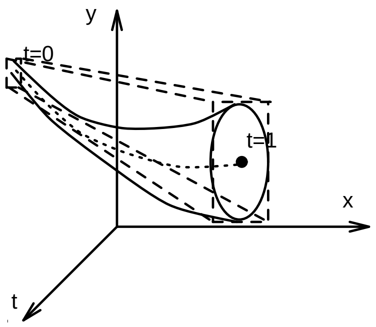


Figure 2.9. A space-time bounding parabolic horn (solid line) when an upper bound on acceleration is known, shown here only for (2+1)-dimensional space. The horn is further bounded by a 4D trapezoid (dashed line).

hierarchy and intersect those first in order to determine the parts which have to be considered further [YW93].

The algorithm for simultaneous traversal of two bounding volume hierarchies is quite similar to a box-tree traversal (see Section 3.6.1):

```
a,b leaves    ⟶     check the parts inside
 a,b disjoint  ⟶     ↑
 a leaf  ⟶
     i = 1...n:
        check  a and  b.i
b leaf  ⟶
     i = 1...n:
        check  b and  a.i
a,b not leaves   ⟶
     i = 1...n:
        a.i intersects  b   ⟶
             i = 1...n:
                check  a.i , b.i
```

Object subdivision. Galonzo's

2.7 Space-Time Approach

The idea with this approach is to consider a dynamic environment in 4 dimensions: space and time [Hubbard93]. Thus we can bound objects not just by volumes in space but also in time *if* their velocity is known. If we also know an upper bound on the acceleration, i.e., $|\ddot{x}(t)| \leq M, 0 \leq t \leq \bar{t}$, then we can bound objects by a parabolic horn in space time (see Figure 2.9).

These horns are further bounded by axis-aligned 4-dimensional trapezoids.

If also the direction of the acceleration is known, then these space-time bounding volumes can be further bounded (or, cut) by hyper-planes given by $\ddot{x}(t) \cdot d \leq 0$.

With these space-time bounding volumes we can compute for any given pair of objects the earliest time where they might collide, because they can't collide as long as their bounding volumes do not collide. Thus the algorithm first calculates the earliest possible time of collision for each pair of objects. Whenever the application issues a collision query for the collision module, the module checks the time and computes only exact collisions for pairs if their earliest possible time is past (the current time has to be announced to the collision module by the application).

2.8 Flexible Objects

Collision detection for flexible objects is needed for physically based simulation and for animation. Typical objects are "soft" objects like clouds, clothes, drops, etc.

Flexible objects are treated by [MW88]. They assume an object to be triangulated. Again, the basic algorithm is a pairwise test of all vertices versus all polygons.

More precisely: each vertex of the moving (deforming) object follows a certain trajectory; these trajectories have to be tested whether or not they intersect with a triangle of the other object. A correct test would have to test also the surfaces generated by moving edges against the triangles of the obstacle. However, [MW88] claims that testing only vertices is precise enough for his purposes (physically based simulation).

Let's assume the moving object M is given at two time steps t and t'; furthermore, let the trajectory of each vertex be linear between t and t'. First, we'll consider the case where the obstacle O is stationary. A vertex P went through a triangle R_0, R_1, R_2 iff

$$P(t) + (P(t') - P(t))t^* = R_0 + (R_1 - R_0)u + (R_2 - R_0)v$$

has a solution (t^*, u, v) with $t^* \in [0,1]$, $u \geq 0$, $v \geq 0$, and $u + v \leq 1$. This is a 3×4 linear equation system which can be solved by the usual techniques (e.g., Cramer, Gauss, Gauss-Seidel).

The hard case (where the obstacle O moves, too) can be tackled by solving the equation

$$P + Vt = R_0 + V_0t + ((R_1 + V_1t) - (R_0 + V_0t))u + ((R_2 + V_2t) - (R_0 + V_0t))v$$

for t. This equation can be expanded to a 5'th degree polynomial in t, which can be solved by standard numerical root finding techniques (Newton, better: binary search, because more robust).

A significant speed up can be obtained by some trivial pre-checks:

- if a point lies on the same side of a polygon at both time steps, then it cannot have gone through that polygon.
- at least one of the end point or start point of a vertex trajectory must lie inside the bounding box of a triangle
- for a given vertex, we want to know quickly all triangles of the obstacle whose bounding box contains, say, the start point. This is a range searching problem, which can be solved quickly by using a point octree [PS85].

An entirely different approach is pursued by [Gascuel93]. He uses a different model for representing flexible objects: they are modeled by a *skeleton* and certain *field functions* f_i; then, the object is defined by

$$P \in \mathbb{R}^3 \text{ for which } f(P) = \sum_{i=1}^{n} f_i(P) = 1$$

Thus, a point P can be defined "inside" or "outside" iff $f(P) > 1$, or $f(P) < 1$, resp.

Testing for collisions is done by choosing a certain amount of sampling points on the surface of every object. (These points stay the same during the whole animation — they are just moved as the surface is deformed.) Then, the inside/outside-function of an object A is evaluated for all sampling points of object B. If any of these is "inside", a collision has occurred.

This scheme can be sped up quite a bit by evaluating first those points which have been deepest inside the other object last time. To save some memory, we could just cache that deepest point with each object pair; for the next frame, we would test this point first, then test points in the neighborhood (this is another example of utilization of *temporal coherence*).

[SWF+93] use conventional parametric or implicit curved surfaces and an interval Newton method to find the minimum of a certain function, which describes the conditions for a collision (see below).

Let the two objects be given by parametric surfaces $S_1(u_1, v_1, t)$ and $S_2(u_2, v_2, t)$. In order to be able to simulate collisions accurately, [Gascuel93] establishes three conditions for a incoming contact collision (see Section 3.1 for an illustration). First, the two surfaces should have some points in common:

$$S_1(u_1, v_1, t) - S_2(u_2, v_2, t) = 0$$

Second, the normal of S_1 has to be perpendicular to S_2 at those collision points, and vice versa:

$$\begin{pmatrix} \frac{\partial S_1}{\partial u_1}(u_1, v_1, t) \cdot N_2(u_2, v_2, t) \\ \frac{\partial S_1}{\partial v_1}(u_1, v_1, t) \cdot N_2(u_2, v_2, t) \end{pmatrix} = 0$$

Finally, an incoming collision satisfies the constraint that the relative velocities of the collision point are directed in the same way as the surface normals at that point.

Altogether, an incoming contact collision can be formulated as a constrained minimization problem

$$\min\{\, t \mid C(u_1, v_1, u_2, v_2, t) = 0\,,\ D(u_1, v_1, u_2, v_2, t) \geq 0 \,\}$$

A similar formulation can be found for implicit surfaces.

An interval Newton method is used to find (several) solutions of the above minimization problem.

In order to reduce the time interval over which solutions are searched, objects are enclosed in spheres big enough to contain the object at all times. The minimization algorithm is used only for the time interval where the two spheres overlap.

For finding *self-intersections* of a polygonal representation of cloth, [VMT94] exploit spatial coherency of the polygon mesh, which is usually a fine discretization of the actual "soft" object.

Two properties are established. The first gives a criterion for non-self-intersection of a contiguous area of the model: if we can find a "normal" such that

1. all dot products of that "normal" with every polygon's normal in that area are positive, and
2. the 2D projection along the "normal" of the border of that area does not intersect itself,

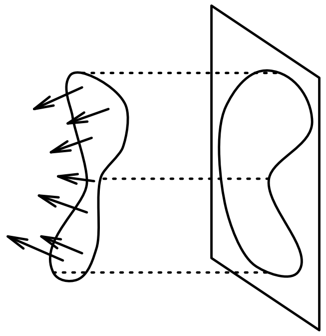


Figure 2.10. A condition establishing non-self-intersection within an contiguous area of a polygonal object.

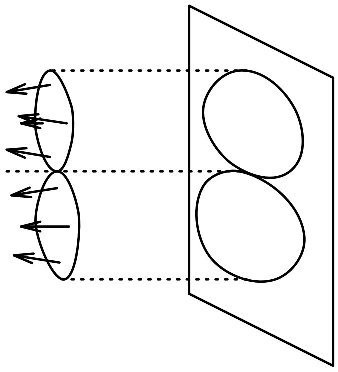


Figure 2.11. A condition establishing non-intersection between two areas of a polygonal object.

then this area is free of self-intersections (see Figure 2.10).

Secondly, a condition for non-collision between two adjacent surface areas (at least one vertex in common) is established: if there is a "normal" such that

1. all dot products of that "normal" with every polygon's normal of both areas are positive, and
2. the 2D projection along the "normal" of the border of the two areas do not intersect each other,

then the two areas do not intersect each other (see Figure 2.11).

2.9 Computing the Exact Time of Collision

An approach which computes the exact time of collision was given by [Canny86]. The method described there can deal with any object topology; the motion of the moving object can be simultaneous translation and rotation. It is convenient that the motion be linear. Obstacles must be stationary.

The main idea which lead to the solution of the problem is to use quaternions to represent orientation. Then, the problem can be formulated in 7-space.

A rotation of a point about the axis $\boldsymbol{n}$ through an angle of θ can be expressed by the quaternion $q = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\boldsymbol{n}$. The rotated point is given by $\boldsymbol{v}' = q\boldsymbol{v}q^*$, where q^* is the multiplicative inverse of q.

A certain position and orientation is a point in *configuration space*. All obstacles can be (conceptually) mapped to *configuration obstacles* (in general curved surfaces), which the object point should not inter-penetrate.

Given two objects A and B, there are three types of contact collisions: 1) a vertex of A touches a face of B, a vertex of B touches a face of A, and an edge of A touches an edge of B (see Figure 2.12).

For every pair of features (of A and B) that can possibly come into contact there will be a constraint. All constraints will define an implicit surface in 7-space which the object point must not penetrate on its path.

The constraint for type . . . is:

1. $[qf_aq^*(p_b - x)] = 0$, i.e., the vertex p_b of B must lie in the rotated plane of A;
2. $[f_b(qp_aq^* + x - p_b)] = 0$;

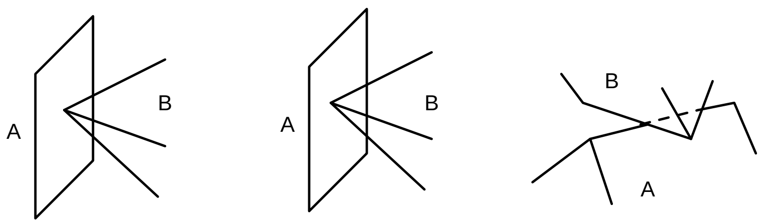


Figure 2.12. The three different types of collision: (a) vertex of A touches face of B, (b) vertex of B touches face of A, (c) edge of A touches edge of B.

3. two edges e_a,e_b of A,B, resp., are in contact, iff $(qp_aq^* - p_b) \cdot qe_aq^* \cdot e_b = 0$

where x and q represent the configuration of the moving object (i.e., its configuration).

All equations above are polynomials in x and q. If a path is given over time, then all constraints together are a univariate polynomial. If, in addition, the path is piecewise linear, then these constraints will become univariate cubics which can be solved by numerical root finding algorithms.

As a by-product, [MW88] get the exact time of collision if it is within the next time step, provided the motion of vertices between two time steps is linear.

2.10 Discussion

As this section shows, considerable effort has been put into research on collision detection. To my knowledge, however, most of the algorithms presented so far are meant to be used with animation (in particular, physically based modeling), path planning for robotics, and computational geometry. Only very few of them (e.g., [LC92, Hubbard93] are really concerned about algorithms allowing interactive collision detection.

Most of the algorithms presented for physically based simulation are too slow for interactive visualization systems, in my opinion. The few results which are presented in the literature seem to confirm that.

Approximate algorithms (e.g., [YK86]) don't seem to be sufficient in the long term, because whenever any kind of realistic object "behavior" is desired, the application probably needs better information than just "*two objects do almost collide*".

Algorithms presented in the area of computational geometry are rather efficient asymptotically; however, I doubt that they are efficient for complexities currently dealt with in interactive visualization systems (100-10000 polygons).

Interesting approaches are those which cache some information obtained at the frame before (e.g., [LC92, Gascuel93]) to be used for the next frame, thus exploiting temporal coherency which is usually given with interactive visualization systems.

The idea of using sphere trees does not appeal too much to me. The problem with spheres is that they usually have to overlap very much in order to cover all polygons. Consequently, if a sphere tree is used as an approximate representation of polyhedra, it is either very coarse, or there are very many spheres. Furthermore, if the sphere tree is a divide-&-conquer data structure for an exact representation, then many polygons are in several spheres at the same time, which causes many polygons being tested more often than necessary. For further discussion of this issue, see Section 3.6.6.

For collision avoidance systems, the approach of [YK86] seems the fastest to me, *if* the environment is otherwise static. But since we are neither interested in collision avoidance alone, nor in almost completely static environments, this approach has not been pursued any further.

Alternate representations like octrees and BSPs seem to allow efficient algorithms for the intersection *construction* problem. However, the *detection* problem is as fast/slow as the *construction* problem. This renders them less attractive for interactive systems since we are only interested in the detection, and maybe in a witness thereof.

For a discussion of bounding volume hierarchies, see Section 5.7.

Chapter 3

Pairwise Collision Detection

3.1 Introduction

This chapter will describe several algorithms for pairwise collision detection, i.e., they solve the following *decision* problem: *given two polyhedra P and Q — do P and Q intersect?* They do not try to solve the *construction* problem of actually generating another polyhedron which is the intersection of P and Q.

Different restrictions on the input objects actually imply (or allow) different definitions of the term *collision*, for example:

- arbitrary objects: $\exists\, edge\ e\, \exists\, polygon\ p :\ e \cap p = x \in \mathbb{R}^3$
- closed objects:
  $$\exists x \in \mathbb{R}^3 :\ x \in P \wedge x \in Q$$
  or:
  $$d(P,Q) := min\{\, |p-q| \ :\ p \in P, q \in Q\,\} > 0$$
  i.e., their distance does not vanish.
- convex objects:
  $$\forall\, edge\ e :\ e \cap P \neq \varnothing$$
  or:
  $$\exists\, plane\ w \in \mathbb{R}^4 :\ \forall\, v \in P :\ v \text{ left of } w\ ,\ \forall\, v \in Q :\ v \text{ right of } w$$
  (there are other definitions possible)

(a) contact collision

(b) proper collision

Figure 3.1. For physically based modeling, we are interested only in contact collisions, because at that particular time new constraints have to be added to the system of constraints.

(a) incoming collision

(b) outgoing collision

Figure 3.2. For physically based modeling, we are interested only in incoming collisions.

In the area of physically based modeling, different types of collisions have to be distinguished (see Figures 3.1 and 3.2). We distinguish between *contact* or *tangent* collisions and *proper collisions*, the former being the ones which we are actually interested in. Furthermore, we distinguish between *incoming* and *outgoing* collisions; doing physically based simulation, we are interested only in the former type, because this is the case where new constraints have to be created.

Algorithms presented in Sections 3.2, 3.4, 3.5, 3.6 have been implemented and evaluated.

3.2 Arbitrary Objects

Every algorithm in this section takes a pair of *arbitrary polyhedra* as input, which means in this context just a collection of plane polygons. With this class of polyhedra, collision cannot be defined anything else than "two polygons intersect". (Two polygons intersect if and only if an edge of one of them intersects with the other.)

Polygons must not be twisted, which is trivially true for triangles. If polygons are not plane, the notion of a polygon normal becomes theoretically void. The Y system still computes it by using Euler's formula [Kirk92a, p. 231]. Let $\boldsymbol{n} = (n_x, n_y, n_z)$ be a polygon normal. For triangles, n_x is exactly the signed area of the triangle projected onto the y-z coordinate plane. Analogous assertions hold for n_y and n_z. If we compute the normal of a non-planar polygon by

$$
\begin{aligned}
n_x &= \frac{1}{2}\sum_{i=0}^{k-1}(y_{i\oplus 1} - y_i)(z_i + z_{i\oplus 1}) \\
n_y &= \frac{1}{2}\sum_{i=0}^{k-1}(x_{i\oplus 1} - x_i)(z_i + z_{i\oplus 1}) \\
n_z &= \frac{1}{2}\sum_{i=0}^{k-1}(x_{i\oplus 1} - x_i)(y_i + y_{i\oplus 1})
\end{aligned}
$$

where $\oplus$ denotes "addition modulo k", then $\boldsymbol{n} = (n_x, n_y, n_z)$ is the normal of a "best-fit" plane which goes through

$$P = \frac{1}{n}\sum_{0}^{k-1} v_i$$

where v_i are the vertices of the non-planar polygon. (Sketch of proof: $(y_{i+1} - y_i)(z_i + z_{i+1})$ is the signed area of the trapezoid below the edge (v_i, v_{i+1}), projected onto the y-z plane.)

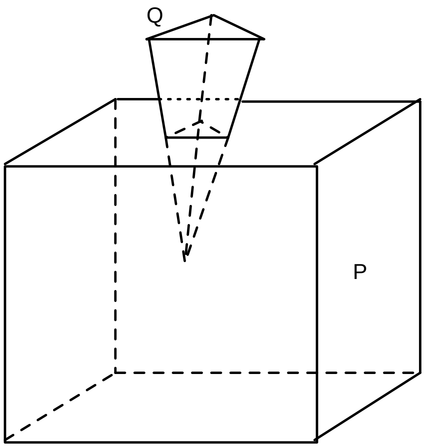


Figure 3.3. A counter-example showing that it is not sufficient to check only edges of P against faces of Q; also, it wouldn't be sufficient to check only vertices of Q for containment in P.

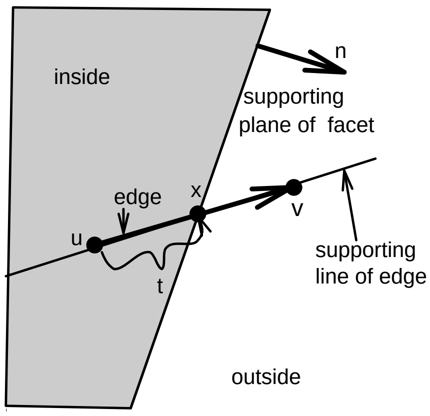


Figure 3.4. Intersection of edge and polygon

Even with non-planar polygons, most of the algorithms in this section can be used "as-is" for an approximate collision detection (depending on the accuracy needed), because they work implicitly with a plane approximation of the polygons, usually represented by a vertex and the plane normal.

The algorithm in this section can handle the largest class of polyhedra: objects which are just a collection of plane polygons. These polyhedra are not really polyhedra in the mathematical sense. An object in this class may even be self-overlapping, and its polygons just should be plane and simple, i.e., the polygons should not be self-overlapping, and the boundary must be one connected path of line segments — they may have holes, though. Thus, an interior can be defined for polygons.

It is highly desirable that a collision detection algorithm can handle this class of polyhedra, since most geometry data, coming from CAD systems, are usually not well-formed objects: there might be gaps between polygons belonging to the same object, polygons could overlap, it might even be unclear which polygons belong to which object!

The basic algorithm presented in this section is "so trivial that it is not even worth a literature reference" [MP78b]. It goes as follows. Check every edge of polyhedron P if it intersects any of the polygons of polyhedron Q, and vice versa. It is not sufficient to check only the edges of P against polygons of Q; besides, it is not sufficient to check whether there are some vertices inside the other polyhedron (if they are closed; see Figure 3.3).

Let P, Q be the two polyhedra to be checked for collision, and let E_P, F_P be the set of edges and polygons, resp. For the time being, we assume that P and Q are given in the same coordinate system (not necessarily the world coordinate system). Let p be an arbitrary vertex of a polygon f with normal n, and let $e = (v, u) \in E^2$ be an edge. Then, the pseudo-code of the algorithm will be as follows:

`Arbitrary collision check`

$\forall e \in E_P \quad \forall f \in F_Q$:
  $\boldsymbol{n} \cdot (\boldsymbol{v} - \boldsymbol{u}) \approx 0 \longrightarrow \quad \uparrow$ `''no collision''`
  $\boldsymbol{n} \cdot (\boldsymbol{v} - \boldsymbol{u}) \not\approx 0 \longrightarrow$
    $t := \frac{\boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{u})}{\boldsymbol{n} \cdot (\boldsymbol{v} - \boldsymbol{u})}$ {*see Figure 3.4* }
    $t \notin [0,1] \longrightarrow \quad \uparrow$ `''no collision''`
    $t \in [0,1] \longrightarrow$
      $\boldsymbol{x} := \boldsymbol{u} + t(\boldsymbol{v} - \boldsymbol{u})$
      $\boldsymbol{x}$ *in polygon* $f \longrightarrow \quad \uparrow$ `''collision''`
      $\boldsymbol{x}$ *not in polygon* $f \longrightarrow \quad \uparrow$ `''no collision''`
$\forall e \in E_Q \quad \forall f \in F_P: \quad \ldots$

Of course, one should not implement this algorithm straight-forward; for details, see Section 7.3.3.

Pre-checks. The algorithm above can be improved by utilizing all kinds of pre-checks.

We observe, that an edge of P can intersect a polygon f only if f is in the bounding volume of P. So, before we check edges of P against polygons of Q, we collect, in a pre-phase, all polygons of Q which are in the bounding volume of P. Then, edges of P are checked only against those polygons of Q which have "survived" this pre-phase.

In order to gain some speed, pre-checks must not be expensive. In this case of collecting polygons, this means, that we can't do an exact check "polygon in bounding volume".

In the following, let's assume that the bounding volumes are boxes (which are the only bounding volumes implemented, for the time being). These considerations apply very similarly to other bounding volumes, too.

This collecting of polygons of Q which are also in the bounding box of P is done by one pass over all polygons of Q. The bounding box is computed for each polygon f by a pass over all its vertices. Then, the bounding box B_f of the f is checked for intersection with the bounding box B_Q of Q. The face will be stored in a list if the two bounding boxes B_Q and B_f overlap.

At first glance, it might seem very inefficient to compute the face bounding boxes from scratch every time the object has moved. One might think that it would be much faster to pre-compute the face bounding boxes and transform those later on. However, in all practical cases, polygons have only 3…5 vertices, and a bounding box can be found by merely comparing points, while transforming a box costs at least 18 multiplications (see Section 5.2.1).

Another very simple pre-check is to test whether the edges e of P are in the bounding box of Q; only in that case, e can intersect with a polygon of Q. There is no need to do this in a pre-phase: since every edge is considered exactly once, this pre-check can be done inside the loop over all edges.

Of course, an efficient implementation does not compute the whole edge bounding box and then check that against the object's bounding box (see Section 7.3.2).

An outline of the algorithm including pre-checks is now as follows:

`Arbitrary collision check with pre-checks`

bounding box of P, Q *not yet valid*

$\longrightarrow$ *calculate bounding boxes of* P, Q
collect faces of Q *whose bbox overlaps bbox of* P,
and store them in $\mathcal{C}$
there is no face of Q *whose bbox overlaps bbox of* P
$\longrightarrow$ $\uparrow$ *no collision*
$\forall e \in E_P$:
e *not in bbox of* Q
$\longrightarrow$ *next edge*
$\forall f \in \mathcal{C}$:
e *not in bbox of* f
$\longrightarrow$ *next edge*
...

3.2.1 Speed-up by Relaxation of Accuracy

As usual, by *weakening demands*, faster algorithms can be found. There are several ways to reduce these demands in the case of collision detection: either by doing only *approximate* collision detection, or by allowing the algorithm to *overlook* some collisions.

Both methods should be available to an interactive system so that the system has the option of choosing less accuracy when the work load becomes too heavy to maintain both, accuracy and interactive frame-rates.

Returning after pre-checks. This is a general technique, which yields approximate collision detection algorithms. Most of them are "collision biased": if they return with "collision", there might still be a chance that there is in fact none; if they return with "no collision", there is none for sure.

The idea is to skip the point-in-polygon test, i.e., when an edge bounding volume overlaps with a polygon's bounding volume, we assume that the edge will intersect with the polygon itself.

This relaxation scheme has not been implemented, yet. The method described next is available.

Allowing the algorithm to miss a collision. If polyhedra are manifold and closed, we can speed up the pre-phase, which collects all polygons Q whose bounding box overlaps with the bounding box P.

The idea is, we don't want to compute face bounding boxes which we discard right after we have computed them. Instead, we want to compute only those polygon bounding boxes which are actually needed; the only problem is: how can we find out whether a polygon's bounding box is needed without actually calculating it?

In order to achieve that, we use the further approximation that a polygon is (partially) inside a given box if one of its vertices is inside this box. This assumption works well for objects which are composed of many small polygons (see, for example, the Y-Potter, Section 8); it fails badly for objects which consist of only a few polygons (like a cube).

The collect phase is a loop over all vertices of object Q; each vertex is tested for containment in P's bounding box. If a vertex $\boldsymbol{v}$ is inside this box, then we will loop over all incident polygons (see Section 4.1.2) and put them in the list (we will call this the *relaxed collect phase*).

Why is the object required to be manifold and closed? Because only in this case, a complete and consistent DCEL can be constructed (see Section 4.1.3). If the object is not manifold, i.e., there are more than two polygons incident to the same edge, then a loop over all polygons incident to one of the edge's end vertices cannot find all polygons. If the object is not closed, then such a loop can't finish at all.

A collision detection algorithm with such a polygon collect phase is biased towards "no collision", i.e., it could miss a collision when in fact there has been one. On the other hand, the answer "collision" is always correct.

Results. The scenario for the timing tests below is the following: all tests were done on an SGI Onyx (2× R4400 150 MHz); rendering was switched off, so almost the whole time was spent for collision detection, which includes calculation of transformation matrices, transformation of vertices and normals, and the like. The simple simulation involved ten objects flying around in a cage made up of six boxes ("walls"); whenever two objects collided (or an object with a wall), their rotational and translational speed was reversed. The objects consisted of 120...1450 polygons (apart from the wall boxes, each consisting of 6 polygons), altogether 4121 polygons. Every frame, 104 collision tests were made.

algorithm (pre-checks accumulated)	coll. det. time / frame (msec) average	shortest	longest	avg. speed-up
without any pre-check	many seconds per frame			
with edge vs. object bbox	5000	150	13000	1.0
with edge vs. face bbox	444	107	2570	11.3
with face collect phase	484	111	3220	10.3
with relaxed collect phase	470	110	2800	10.6

I suspect that the relaxed collecting phase doesn't speed up things, because all objects are close to each other. So, eventually, the bounding boxes of all faces have to be computed, anyway. However, it will speed up collision detection a lot with the potter application (see Section 8.1). For a pot of 9944 polygons, and with collision detection with the full hand (16 hand objects), we will get a frame-rate of 5–7 frames/sec when using the relaxed collect phase. On the other hand, if we use the exacter face collect phase, the frame-rate will go down to only 3–4 frames/sec, i.e., in this particular case, the relaxed collect phase yields a speed-up of about 2.

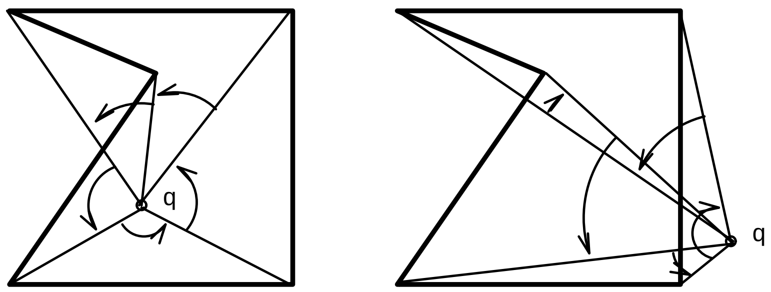
Whether the exact face collecting phase speeds up the overall performance or not seems to depend on the architecture: timing tests showed that it does help on SGI's Skywriter (8 R3000, 40 MHz), but not on Onyx and Indigos.

The pre-check "object bounding box vs. object bounding box" is not taken into consideration here, because this is something which will be done on the global collision detection level, above the pairwise object level presented in this chapter.

3.2.2 Point-in-Polygon Test

At the heart of the algorithm sits the test whether a given point $\boldsymbol{p}$ is inside a given polygon f, both in $\mathbb{R}^3$. The polygon may be non-convex (it must be plane, though). We will assume in this section that the point $\boldsymbol{p}$ is located somewhere in the supporting plane of the polygon f.

Since this problem is *two-dimensional* in nature, we will reduce the input data onto 2-space. We will do this by *projecting* both point and polygon onto one of the coordinate planes, by throwing away one coordinate. We cannot, however, project the data always onto the same plane, because this could produce numerically bad conditioned data (theoretically, the polygon might even degenerate)! To avoid that,



(a) sum of angles is 2π (b) sum of angles is 0

Figure 3.5. Point-in-polygon test by the sum of angles method

we first find that coordinate plane among x-y, x-z, and y-z plane, which is "most parallel" to the polygon. This plane can be identified easily by just finding that element of the polygon's normal $\boldsymbol{n} = (n_x, n_y, n_z)$ which is largest by absolute value, i.e., if n_x is the largest component of $\boldsymbol{n}$, then the y-z plane will be "most parallel" to the polygon.

It is obvious that $\boldsymbol{p} \in f \Leftrightarrow \pi(\boldsymbol{p}) \in \pi(f)$, where $\pi(x)$ is the projection of x onto the coordinate plane.

A real implementation, of course, would *not* really perform the projection (that would involve copying the data)! Instead, only two variables are determined, which are used later on as indices into the three-dimensional data.

Henceforth, we will consider all data to be given in the x-y plane, without loss of generality. We will denote the vertices of the polygon P by $p_i, i \in [0, n-1]$, and the query point by q.

There are many algorithms to determine the containment of a point in a polygon [Glassner90a, PS85, Glassner89, Hay92] .

Sum of angles. Let $d_i = p_i - q$, $\measuredangle(d_i, d_{i+1}) = \arccos \frac{d_i \cdot d_{i+1}}{|d_i| \cdot |d_{i+1}|}$, $d_n = d_0$. Then

$$q \in f \quad \Longleftrightarrow \quad \sum_{i=0}^{n-1} \measuredangle(d_i, d_{i+1}) \;=\; 2\pi$$

otherwise, $\sum = 0$ (see Figure 3.5). The arccos can be used, actually, since angles can't get larger than 180°. Of course, a real implementation would have to test for $\sum > 0$, because of ubiquitous round-off errors.

For a fast implementation, the arccos should be calculated by look-up tables; on the other hand, the more vertices a polygon consists of, the more accurate this look-up table has to be.

One-shot. The idea is to shoot a ray emanating from q in an arbitrary direction and to count the number of intersections of the ray with the edges of the polygon. If the number of intersections is odd, then q is inside the polygon. Since the ray can be any one, it is most efficient to choose a special one, say, along the +x axis.

It turns out that most of the intersections actually don't have to be computed at all, but can be decided by simple coordinate comparisons. Let $p_i = (x_i, y_i)$ be the vertices of the polygon, and $q = (\bar{x}, \bar{y})$ the query point. Then the algorithm is as follows:

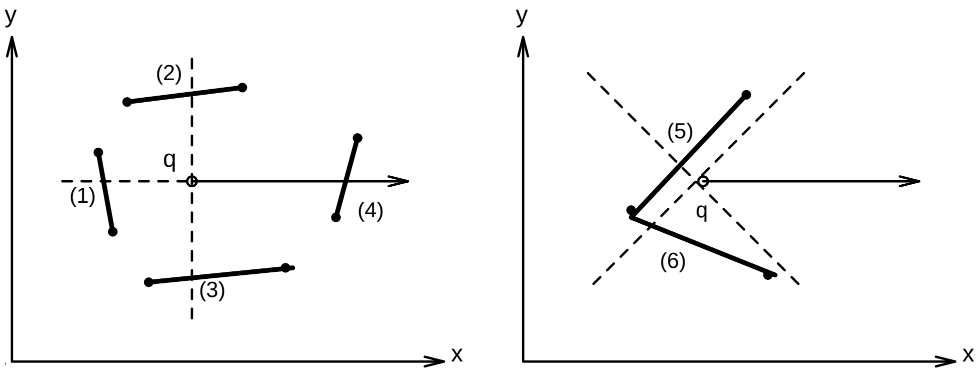


Figure 3.6. The six possible pre-checks with the one-shot method for the point in polygon test.

`Point-in-polygon test`

$i = 0 \ldots n-1$:

$x_i \leq \bar{x} \ \wedge \ x_{i+1} \leq \bar{x}$
$\longrightarrow$ *no intersection* {*Fig. 3.6, case (1)* }

$y_i > \bar{y} \ \wedge \ y_{i+1} > \bar{y}$
$\longrightarrow$ *no intersection* {*Fig. 3.6, case (2)* }

$y_i \leq \bar{y} \ \wedge \ y_{i+1} \leq \bar{y}$
$\longrightarrow$ *no intersection* {*Fig. 3.6, case (3)* }

$x_i > \bar{x} \ \wedge \ x_{i+1} > \bar{x}$
$\longrightarrow$ *intersection* {*Fig. 3.6, case (4)* }

$(y_i - \bar{y}) \geq (x_i - \bar{x}) \ \wedge \ (y_{i+1} - \bar{y}) \geq (x_{i+1} - \bar{x})$
$\longrightarrow$ *no intersection* {*Fig. 3.6, case (5)* }

$-(y_i - \bar{y}) \geq (x_i - \bar{x}) \ \wedge \ -(y_{i+1} - \bar{y}) \geq (x_{i+1} - \bar{x})$
$\longrightarrow$ *no intersection* {*Fig. 3.6, case (6)* }

$t = \frac{(y_i - y_{i+1})(x_i - \bar{x}) - (x_i - x_{i+1})(y_i - \bar{y})}{y_i - y_{i+1}}$ {*don't calculate the division!* }

$t \geq 0 \longrightarrow$ *intersection*

Why did we use "$\leq$" and not "$<$"? Normally, we would have to deal with the special case that a vertex lies exactly on the ray[1]. If that happened, we could always find an $\varepsilon > 0$ and $U_\varepsilon(q)$ such that $U_\varepsilon(q)$ would be completely inside or outside the polygon (depending on q), and such that the ray $(\bar{x} + \varepsilon, \bar{y}) + t(1, 0)$ would not hit any vertex of the polygon anymore. (Because there are only finitely many vertices.) This offsetting of q by ε is implicitly done by the use of "$\leq$" and "$>$" instead of "$<$", "$=$", and "$>$".

The test for the sign of t cannot be simplified anymore, because $(y_i - y_{i+1}) < 0 \Leftrightarrow (y_i - \bar{y}) < 0$ (and analogously for x).

Of course, an efficient coding re-uses terms like $(y_{i+1} - \bar{y})$ in the next loop iteration.

Results. Almost all of the ray-edge intersection tests can be decided by the pre-checks $(1) \ldots (6)$:

[1] In 2D we could resolve this case by taking edge normals into account.

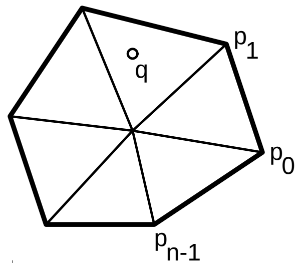


Figure 3.7. A convex polygon can be partitioned into wedges

pre-check	positive
1	50%
2	22%
3	26%
4	1%
5	0.00003%
6	0.00002%
remaining	1%

(Based on about 3 million ray-edge intersection tests.) This shows that for only 1% of all ray-edge intersection tests the line parameter t has really to be computed.

Box-tree for point-in-polygon test. The one-shot algorithm above can be improved by using the box-tree data structure of Section 3.6. That data structure would allow to discard many polygons trivially, not only one at a time as with the pre-checks from above.

3.3 Objects Consisting of Convex Polygons

The algorithm above for testing whether a point is inside a polygon still has asymptotical complexity $O(n)$ (n = # vertices). If the objects to be tested for collision are made of convex polygons, then we can use an asymptotically more efficient algorithm for the point-in-polygon test; thus the overall collision algorithm will be asymptotically more efficient.

Binary search. Let f be a polygon in the plane, $p_0 = (x_0, y_0), \ldots, p_{n-1} = (x_{n-1}, y_{n-1})$ be its vertices (for convenience, we identify p_n with p_0). Since the polygon f is convex, we can partition it into *wedges*. Each wedge w_i is the triangle formed by the barycenter of the polygon and vertices p_i and p_{i+1} [PS85] (see Figure 3.7). Each vertex p_i can be represented in polar coordinates (θ_i, r_i), with the barycenter being the origin; here, we are interested only in θ. Likewise, the query point $q = (\bar{x}, \bar{y})$ can be represented by $(\bar{\theta}, \bar{r})$. When we have found two vertices p_i, p_{i+1} with angles θ_i and θ_{i+1} such that $\theta_i \leq \bar{\theta} \leq \theta_{i+1}$, then we have to check only whether $\begin{pmatrix} y_{i+1} - y_i \\ x_i - x_{i+1} \end{pmatrix} (q - p_i) \geq 0$, in order to find out if q inside f. (This last check takes $O(1)$ time.)

Given a query point q, we can find the wedge w_i, in which it is located, by *binary search* in the polar angles of the vertices of f.

Actually, this method works for the larger class of star-shaped polygons, too.

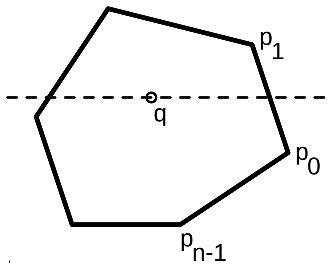


Figure 3.8. The one-shot method for convex polygons; exactly two edges are hit by the line, or none at all.

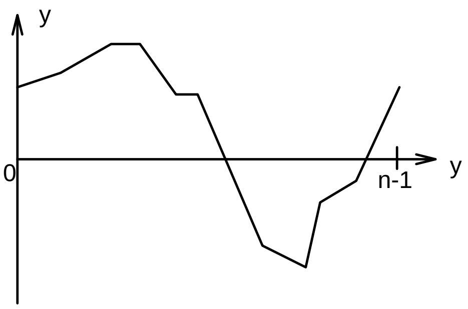


Figure 3.9. A bitonic sequence, if cyclically continued.

The application to moving objects (and hence, moving polygons) in 3D adds a little problem. In the 2D case above, we assumed that the vertices are stored sorted by angle. This sorting order gets disturbed a little bit when the polygon is moved. So, we have to transform the query point q back into the coordinate system in which the vertices have been sorted.

We do not necessarily have to store the angles together with the vertices; we can as well compute them "on-the-fly".

We can do better, however, by combining binary search with the one-shot method.

Binary search and one-shot. Since the polygon f is assumed to be convex, a horizontal line through a query point q will hit the border of f exactly twice or not at all (see Figure 3.8). If the line doesn't hit the border at all, q is outside; if it intersects with two edges (and we know them), we just have to check whether the two intersection points are on opposite sides of q or on the same side.

The idea is to find the two edges (if at all) by a search similar to binary search, called *Fibonacci search* (see [PFTV88]). This technique is suitable for searching a bitonic sequence for a global minimum. A *bitonic sequence* has exactly one range where it is monotonically increasing, and one range where it is monotonically decreasing. We consider the sequence of y-coordinates of the vertices of f; if we consider them cyclically, they will form a bitonic sequence (see Figure 3.9).

First, we'll solve a little sub-task: given vertices p_i and p_j $(i \neq j)$ with $y_i > \bar{y}$ and $y_j < \bar{y}$; thus, there is exactly one edge (p_k, p_{k+1}), $i \leq k < j$, which intersects the line $y = \bar{y}$. (Still, we assume, without loss of generality, that the line $y = \bar{y}$ doesn't contain any vertex of f; see Section 3.2.2.) The algorithm is quite simple:

`while` $|i - j| > 1$ $\quad\quad$ $\{$*not necessarily* $i \leq j$ $\}$
$\quad k := \frac{i+j}{2}$
$\quad y_k > \bar{y} \longrightarrow \quad i := k$
$\quad y_k \leq \bar{y} \longrightarrow \quad j := k$

Afterwards, (p_i, p_j) will be the edge which intersects the line $y = \bar{y}$.

Now we can tackle the task of finding the two edges which intersect $y = \bar{y}$, or finding out that the entire polygon is above or below this line.

<u>`One-shot with Fibonacci search`</u>

$i := 0, \quad j := n - 1$
w.l.o.g. $y_i < y_j$ $\quad\quad$ $\{$*else swap* i, j $\}$

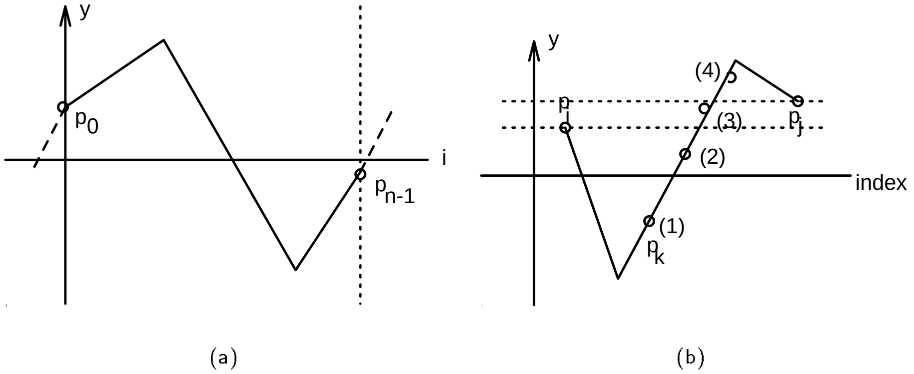


Figure 3.10. The possible cases with the Fibonacci search in the one-shot point-in-polygon test

$(y_i > \bar{y} \wedge y_j < \bar{y}) \vee (y_i < \bar{y} \wedge y_j > \bar{y})$
$\longrightarrow$ *{Fig. 3.10(a) }*
edge (p_j, p_i) *intersects with the line* $y = \bar{y}$,
there is exactly one other edge which does so, too,
find that one with the sub-program above
$y_i > \bar{y} \wedge y_j > \bar{y}$
$\longrightarrow$ *{Fig. 3.10(b) }*
try to find k *so that* $y_k < \bar{y}$ *or* $y_k = \min\{y_l\}$
`while` $|i - j| > 1$
$k := \frac{i+j}{2}$
$y_k \leq \bar{y}$
$\longrightarrow$ *{Fig. 3.10(b), case (1) }*
find the two intersecting edges between p_i *and* p_k,
and between p_j *and* p_k *using the sub-program above,*
then exit
$y_k < y_i$ $\{y_k < y_i \Rightarrow y_k < y_j\}$
$\longrightarrow$ *{Fig. 3.10(b), case (2) }*
$j := i,\ i := k$
$y_i < y_k \wedge y_k < y_j$
$\longrightarrow$ *{Fig. 3.10(b), case (3) }*
$j := k$
$y_j < y_k$ $\{y_j < y_k \Rightarrow y_i < y_k\}$
$j := k$
if `while` *runs through* $\longrightarrow$ f *completely above* $y = \bar{y}$
$y_i < \bar{y} \wedge y_j < \bar{y}$
$\longrightarrow$ ...

The loop invariant is $\{\ y_i < y_j\ \}$, the loop variant is $\{\ |i - j|\ searrow\ \}$.

Precisely, $|i' - j'| \leq \frac{1}{2}|i - j|$ with every loop iteration. Thus, the overall complexity is $O(\log n)$, $n = $#vertices. Plus, no pre-computation is necessary.

The algorithm above has not been implemented, though, because I don't think that practical applications of any collision detection will involve objects containing many polygons with more than a few (3–10)vertices; even if it contained a few polygons with some 10 to 100 vertices, we still think that the hidden constant within the above algorithm would offset the gain.

3.4 Convex Objects

The restriction of a geometric problem to convex polyhedra quite often results in more efficient algorithms. The same effect could be expected with collision detection.

Convex polyhedra also give rise to several different ways to regard the problem than arbitrary polyhedra could offer: one can consider a convex polyhedron not only as a collection of polygons, edges, and vertices with certain properties — but also as the intersection of half spaces, or as the convex hull of its vertices. All of these representations are, in fact, merely different ways to *look* at the data, because with each way, all the data (and maybe more) are already there within a b-rep.

These different representations lead to different approaches.

In this section, whenever the term polyhedron is used, we actually mean *convex polyhedron*.

3.4.1 A Modified Cyrus-Beck Algorithm

Given two convex polyhedra P and Q, the idea is to *clip P by Q* and vice versa. (As in Section 3.2, it is not sufficient to clip only P by Q, see Figure 3.3.) Since both of them are convex, the Cyrus-Beck algorithm can be used [FvDFH90], which will be modified to take advantage of the special situation.

The *Cyrus-Beck algorithm* works as follows: The polyhedron is represented by the intersection of half spaces, which are defined by the polygons of the object. An edge is represented by its parametric form.

$$\begin{aligned} H &: (x-p)n \leq 0 \\ e &: x = u + t(v-u) \end{aligned}$$

where n is the normal of the polygon, pointing to the "outside", p is an arbitrary point on the supporting plane of the polygon (e.g., the first vertex), u and v are the vertices of the edge, $t \in [0,1]$.

By clipping the edge e at H, the permissible interval of t is restricted; we never have to compute any points. If e is completely inside H, the interval for t will stay the same (it cannot get larger). Whenever the permissible interval for t becomes $\varnothing$, the edge is completely outside the polyhedron.

In the following, we need the notion of *entering* or *leaving* edges. An edge $e = (u,v)$ *enters* a half space if $(v-u)\cdot n < 0$ (this means that we go from "outside" to "inside" with increasing values of t); in this case, the permissible interval $[\alpha,\beta]$ for t would be restricted from above, if at all, i.e., β would be replaced by $\beta' < \beta$. The edge *leaves* if $(v-u)\cdot n > 0$, and if t got restricted, α would be replaced by $\alpha' < \alpha$.

The supporting line of the edge e intersects with the boundary of H at

$$t^* = \frac{n\cdot(p-u)}{n\cdot(v-u)}$$

if $n\cdot(v-u) \neq 0$.

With clipping, the special case where e is parallel to the boundary of H, i.e., $n\cdot(v-u) = 0$, has to be taken care of. With collision detection of convex polyhedra, we don't have to do that; instead, we can just ignore this case and proceed to the next half space. The reason is: if the two polyhedra do intersect, then there will be another edge (either of P or Q), which actually does intersect a polygon; if the two do not intersect, no harm is done by ignoring the parallel cases.

A simple coding of the algorithm would look like:

<u>`Cyrus-Beck collision detection`</u>

$\forall e = (u, v) \in E_P$
  $[\alpha, \beta] := [0, 1]$
  $\forall f \in F_Q$:
    $t^* = \frac{n\cdot(p-u)}{n\cdot(v-u)}$
    $n \cdot (v - u) < 0$
    $\longrightarrow$ $\{[-\infty, t^*]$*-segment of line is outside* $\}$
      $\alpha := \max\{\alpha, t^*\}$
    $n \cdot (v - u) > 0$
    $\longrightarrow$ $\{[t^*, \infty]$*-segment of line is outside* $\}$
      $\beta := \min\{\beta, t^*\}$
    $n \cdot (v - u) \approx 0$
    $\longrightarrow$ *do nothing*
    $\alpha > \beta$ $\{$*edge completely outside* $\}$
    $\longrightarrow$ *next edge*
  $\alpha \le \beta$ $\{$*edge (partially) inside* $\}$
  $\longrightarrow$ $\uparrow$ *P and Q do intersect*
do the same again with P and Q swapped

Complexity. Euler's formula for convex polyhedra tells us, that $O(|V|) = O(|E|) = O(|F|)$. The clipping of an edge by one polygon takes $O(1)$ time; thus, the above algorithm takes $O(n^2)$ time, where $n = |V_P|$ (we assume $|V_P| \approx |V_Q|$).

Reducing the number of edge clippings. As with arbitrary polyhedra, we can reduce the computational work by a pre-phase. Of course, we can check an edge against the other object's bounding box. Also, we can use the face-collecting pre-phase, which collects all those polygons of Q being (partially) in P's bounding box; we will call this set of polygons $\mathcal{C}$. Then, we need to clip edges of P only by polygons in $\mathcal{C}$.

Why are we allowed to "forget" about certain polygons when clipping edges? When an edge e of P is clipped by the polygons in $\mathcal{C}$, three cases can occur. First: $\mathcal{C}$ is empty; then e is outside Q. Second: e is completely clipped away by the polygons in $\mathcal{C}$; then e is completely outside Q (it cannot be (partially) inside $Q \setminus \text{bbox}(P)$). Third: e is not completely clipped away by the polygons in $\mathcal{C}$; then e is (partially) inside Q. It cannot be clipped away any further by the "forgotten" polygons outside the bounding box of P, because Q is convex.

Unfortunately, we cannot trivially abort a clipping operation if the edge is not (partially) inside the face's bbox, since we clip edges always by whole half-spaces.

Results. A few timings were done to evaluate the efficiency of restricting the class of polyhedra to convex ones. The following table shows the improvements of the various pre-checks:

pre-check	speed-up
none	1.0
edge vs. obj. bbox	1.3
edge vs. face bbox	1.8

Pre-checks are accumulative, and speed-up is with respect to the run-time of the algorithm using one pre-check less.

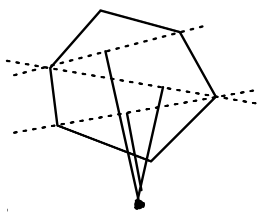


Figure 3.11. Pulling a plane, supported by three vertices of the polyhedron, towards the point, skipping many vertices every step might yield an efficient algorithm to compute the distance between a point and a polyhedron.

A comparison with the arbitrary algorithm was carried out on an Onyx (150 MHz) and a VGX (40 MHz). Only two objects moved in a cage of size 1^3, the objects having a size of 2^3; so, almost every frame a collision occurred. The following table shows the results for the Onyx; each figure is the average frames/sec on 10,000 frames, no rendering was done:

algorithm	frames/sec per complexity			obj.-type
	2×250	2×120	2×60	
arbitrary	6.7	2.4	1.1	spheres
convex	7.4	2.1	2.5	
arbitrary	7.4	2.1	0.8	cones
convex	6.2	2.3	1.1	

Tests on the VGX showed similar results (just scaled by the clock frequency).

These results astonished me: one would think, that the convex algorithm is much faster, because the computation for one edge (tested/clipped against all polygons of Q) seems to be much less! The reason might be, that the arbitrary algorithm can discard much more computation by simple pre-checks. (I am still somewhat suspicious, especially because at a (rather early, though) time the convex algorithm indeed seemed to be faster...)

Future directions. Inspired by [GJK88], [LC91] the algorithm sketched in the following might yield significant speed-up, since it is a divide-&-conquer method.

Goal: compute the distance of a point to a convex polyhedron. Transform the point into the coordinate system of the polyhedron. Start with an arbitrary plane through three vertices of the polyhedron. Pull the plane towards the point by replacing one of the three supporting vertices with one which is nearer to the plane (see Figure 3.11).

If there is no nearer plane such that the point stays on the same side as before, then the point is inside the polyhedron.

If there is no nearer plane, then the point is outside: the closest feature of the polyhedron could be the supporting polygon of the plane, an edge, ora vertex of it.

The vertex replacing step could be done by a divide-&-conquer method: the vertices of the polyhedron had to be stored sorted by x-coordinate (and by y-coordinate and z-coordinate in separate lists).

Beware of cycles. Can't happen if the distance is strictly smaller with every step.

3.4.2 Separating Planes

If we consider a convex polyhedron to be the convex hull of its vertices, we can reformulate the condition for collision detection as follows: P and Q do not intersect if and only if there is a plane h such that all vertices of P are on one side, and all

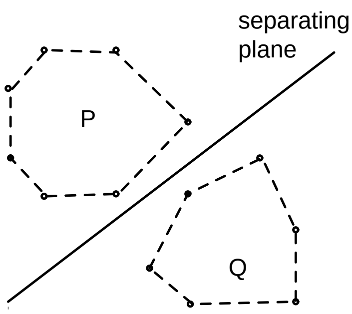


Figure 3.12. Two convex polyhedra can always be separated by a plane.

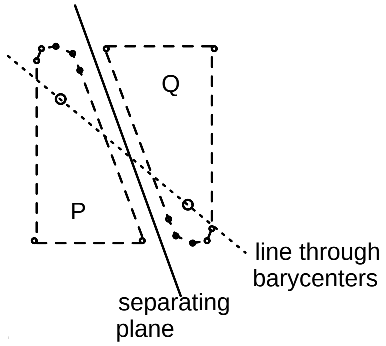


Figure 3.13. The separating plane is not necessarily perpendicular to the line through the two barycenters.

vertices of Q are on the other side (see Figure 3.12). Such a plane will be called a *separating plane*, and P and Q will be called *linearly separable*.

Let $P = \{p^1, \ldots, p^n\}, Q = \{q^1, \ldots, q^m\} \subseteq \mathbb{R}^3$. Then

$$\begin{aligned} &P, Q \text{ are } \textit{linearly separable} \quad :\Leftrightarrow \\ &\exists w \in \mathbb{R}^3, w_0 \in \mathbb{R}\, \forall i : p^i \cdot w - w_0 > 0 \ , \ q^j \cdot w - w_0 < 0 \end{aligned}$$

We translate the above condition into projective space and get

$$\begin{aligned} &P, Q \textit{ linearly separable } \Leftrightarrow \\ &(p^i, -1) \cdot (w, w_0) > 0 \ , \ (q^j, -1) \cdot (w, w_0) < 0 \Leftrightarrow \\ &(p^i, -1) \cdot (w, w_0) > 0 \ , \ (-q^j, 1) \cdot (w, w_0) > 0 \end{aligned}$$

This means that we can always assume, without loss of generality, that the separating plane (if any) goes through the origin.

One might be tempted to think that it would suffice to compute the barycenter of the vertices of P and Q, resp., and then check whether there is a separating plane perpendicular to the line through these two points. A counter-example can be found in Figure 3.13.

The algorithm for finding a separating plane chosen here is inspired by neural networks, strictly speaking, by *perceptrons* (also called "feed-forward networks"; [HKP91]). There exists a very simple algorithm to compute w for perceptrons, which is a kind of a *probabilistic algorithm* (see 3.5), but could be turned into a deterministic algorithm:

<u>`Perceptron learning rule`</u>

`input:` $Z = \{z^k\} := \{(p^i, -1), (-q^j, 1)\}$
guess an arbitrary start vector w^0*, say,* $(0, 0, 0, 1)$
`loop`

$$\begin{aligned} &\exists z : z \cdot w^l < 0 \\ &\longrightarrow \qquad w^{l+1} := w^l + \eta \cdot z \\ &\forall z : z \cdot w^l > 0 \\ &\longrightarrow \qquad \uparrow w \textit{ is separating plane} \end{aligned}$$

The idea of this algorithm is to "turn" the separating plane a little bit whenever we find a point z which is still on the "wrong" side of it (see Figure 3.14).

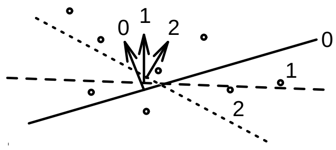


Figure 3.14. Each point found on the "wrong" side pulls the plane normals towards itself.

Termination. If P and Q actually intersect, the algorithm presented above won't terminate! This can be fixed in two possible ways: One way would be to stop the loop after a certain amount of iterations (say 1000), and if it had not found a separating plane, we would just assume that P and Q are not linearly separable. This would turn the algorithm into a *probabilistic* one (see also Section 3.5); if it returned with "not linearly separable", there would be a small chance that the answer is wrong. On the other hand, if it returned with "linearly separable", the answer would always be correct. Thus, the algorithm would be *biased* towards "not linearly separable".

There is another possibility to make the algorithm terminate always: we can try to establish an upper bound on $l_{\max}$, the *maximum number of loop iterations* needed to find a separating plane for two linearly separable polyhedra. Let's assume P and Q are linearly separable. Given a separating plane w, let $D(w) = \frac{1}{|w|}\min\{wz^i\}$ be the distance of the point z which is closest to the plane w. $D(w)$ is the "goodness" of w; $D(w) < 0$ means w is not a separating plane. Let w^* be the "optimal" separating plane (w.l.o.g., $|w^*| = 1$), in the sense that $D_{\max} := D(w^*) = \max\{D(w) | w \in \mathbb{R}^4, |w| = 1\}$. $D_{\max} > 0$ means that P and Q are linearly separable.

As long as the algorithm has not found a separating plane, w^l will be updated by $\eta \cdot z$ (because $z \cdot w^l < 0$). After $l_{\max}$ iterations, $w^{l_{\max}} = \eta \sum_i k_i z^i$, where k_i is the number of times w has been updated by ηz^i, and $\sum k_i = l_{\max}$.

First, we'll show a lower bound on $|w^l|$:

$$\begin{aligned}
& w^l w^* = \eta \sum k_i z^i w^* \geq \eta \sum k_i D(w^*) = \eta D_{\max} \sum k_i \quad \Rightarrow \\
& |w^l|^2 = |w^l|^2 \cdot |w^*|^2 \geq |w^l w^*|^2 = \eta^2 D_{\max}^2 l^2 \quad \Rightarrow \\
& |w^l|^2 \geq \eta^2 D_{\max}^2 l^2
\end{aligned} \tag{3.1}$$

Without loss of generality, we can assume that $\forall i : |z^i| = 1$. Otherwise, we could scale all points z^i such that they are mapped on points inside the unit circle; this would not change the property of linear separability. With this assumption, we can also show an upper bound for $|w^l|$:

$$\begin{aligned}
& |w^{l+1}|^2 - |w^l|^2 \\
= \; & (w^l + \eta z^{i_l})^2 - |w^l|^2 \\
= \; & |w^l|^2 + \eta w^l z^{i_l} + (\eta z^{i_l})^2 - |w^l|^2 \\
= \; & \eta w^l z^{i_l} + (\eta z^{i_l})^2 < 0 + \eta^2 \cdot 1 \quad \Rightarrow
\end{aligned}$$

$$|w^l|^2 - |w^0|^2 < l\eta^2 \tag{3.2}$$

Equations 3.1 and 3.2 yield the desired upper bound on $l_{\max}$:

$$\eta^2 D_{\max}^2 l_{\max} \leq |w^{l_{\max}}|^2 < l_{\max}\eta^2 \quad \Rightarrow$$

$$l_{\max} < \frac{1}{D_{\max}^2} \tag{3.3}$$

This upper bound is independent of the number of vertices; it depends only on the "distance" (given by D) between P and Q.

We could use this upper bound in order to determine when the algorithm should terminate; to do that, we would need to estimate $D_{\max}$, probably on-the-fly, i.e., we could improve an estimate of $D_{\max}$ after each complete pass over all points.

The separating plane algorithm presented so far has several *advantages*: it does not depend on the volume of the intersection of the two bounding boxes of P and Q, because there is no pre-check (yet) depending on that. The greatest advantage is that a *non-collision can be recognized very quickly* (non-collision is the worst case for all other algorithm presented here); depending on the initial (guessed) separating plane, non-collision could be established by a single pass over all points. In any case, non-collision is recognized the faster the further P and Q are apart from each other.

But there are also some *disadvantages*: it is probabilistic, and it does not provide a witness of intersection (i.e., a polygon and an edge). On the other hand, when P and Q do intersect, it can find the two vertices which are furthest "inside" the other object, *if* we modify the algorithm to weigh η by the "badness" with which a point is on the wrong side.

This algorithm can be used for an approximate collision check with *non-convex objects*. This is an example of an implicit alternative representation of non-convex objects: we approximate them by their convex hull, which is the better the more "convex" these objects are.

The algorithm is biased, as the result "not colliding" is always correct, even for non-convex objects.

Because non-collision is the quicker case with this algorithm, whereas collision is the quicker for most of the other algorithms, it could be used as a concurrent dual in a parallelization of the collision detection task (see Section 6.5).

Relaxation of η. The algorithm above is a kind of an optimization algorithm. The goal is to find the global maximum $D_{\max}$. Step by step, the algorithm tries to move further into the direction of that maximum. However, unlike other optimization algorithms, it does not yet reduce the step width η, the further it moves.

There are (at least) two ways to incorporate that reduction of the step width: *constant* and *adaptive*. The constant reduction just reduces η by a constant factor with every update of the separating plane. This has been implemented; a factor of 0.97 turns out to be optimal.

The other way would be to base η itself on the "badness" of the current separating plane with respect to the current point which is on the wrong side. Then, an update would be computed by $w^{l+1} := w^l + \frac{w^l z}{|w^l||z|} \cdot z$. However, it is not clear without actually trying that this approach is really faster, since the update equation is computationally much more expensive than the original one. Which one is the best can only be evaluated by experiments.

Bounding box pre-check. As with previous algorithms, a simple bounding box pre-phase could be added to the algorithm: let B be the intersection of the bounding box of P with the one of Q; then, for all points outside B, a separating plane can be given trivially. On the other hand, if there is a separating plane for the points in B, then this plane is also a separating plane for all the points in P and Q (this is true only for convex polyhedra). Thus, P and Q are linearly separable if and only if the points in B are linearly separable.

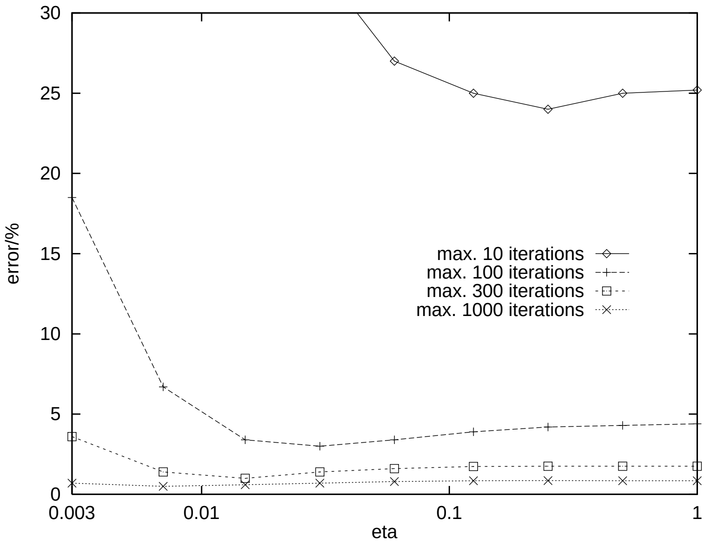


Figure 3.15. Impact of η and the max. number of iterations on the correctness of the separating planes algorithm. The error is the number of times where the separating planes algorithm returned with "collision", but there was none, in fact.

Incremental collision detection. The separating planes algorithm can be easily extended to an incremental method (which has been implemented): for each pair of objects we store the plane which the algorithm ended up with. When the same pair of objects is checked for collision the next time, we use this plane for the initial guess. If objects have not moved very much since the last collision query, this initial guess will probably be a good guess. If the two objects didn't collide the last time, and they don't collide this time, chances are good that one pass over all points will suffice to establish non-collision.

Results. Several tests were carried out to evaluate the feasibility of the approach. Two convex objects (200 polygons in total) were flying in a cage of 6 boxes; two cones were chosen, because these are convex, but can be arranged in a way like in Figure 3.13, which is a hard case for this algorithm. Whenever the two objects collided, they bounced off each other; no collision detection with walls was done. Rendering was switched off. Each figure is an average over 10,000 frames.

The first test was designed to find out the impact of two parameters: the maximum number of iterations, and the update parameter η (no relaxation of η); see Figure 3.15[2].

With the second test, I tried to measure the effect of the method how the initial plane is computed[3]. Three methods were taken into consideration: first, the trivial one ($w = (1, 0, 0, 0)$). Second, an initial plane which goes through the midpoint between the two barycenters of P and Q; the normal of the plane is the line which goes through the two barycenters. The third method of calculating the initial plane is like the second, but using the barycenters of the bounding boxes of P and Q instead of their true barycenters.

[2] Invocation: `movem -x 10 -t 1000 co -e 1.5 -p <eta> <max.iter.>`;
`else` part in `movemCB` in `movem` should be commented in!

[3] Invocation: `movem -x 10 -t 1000 coordinate -e 1.5 -p 0.03 [10 | 100 | 1000]`, with different methods compiled in.

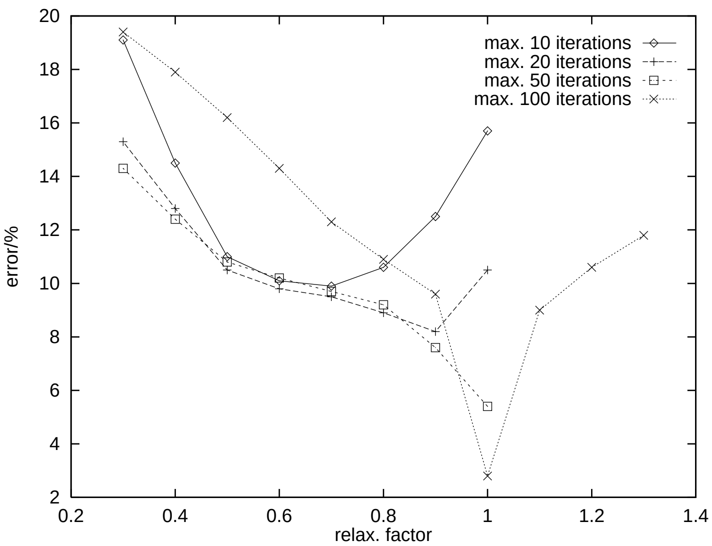


Figure 3.16. Optimum relaxation factor for η with various maximum iterations; the initial η is always the one which proved to be optimal when doing no relaxation. A relaxation factor of 1.0 means no relaxation at all.

initial plane	error/% (max. num. iter.)			improvement		
	iter. ≤ 10	iter. ≤ 100	iter. ≤ 1000			
	$\eta = 0.25$	$\eta = 0.03$	$\eta = 0.007$			
trivial	24	3.0	0.5	1.0	1.0	1.0
barycenters	16	2.8	0.37	1.5	1.07	1.35
bbox barycenters	15	2.8	0.37	1.6	1.07	0.37

This shows that it doesn't matter very much which initial separating planes we use; thus, we could as well use the one that is most inexpensive to calculate (which is $w = (1, 0, 0, 0)$).

The last test's purpose was to find out the optimum factor for the relaxation of η. Again, the test was done for various maximum numbers of iterations; for each of those, the optimum initial η was chosen. The initial guess for the separating plane was done with the third method, i.e., the plane between the barycenters of the bounding boxes. See Figure 3.16 for the results. It is not clear to me why a relaxation factor helps only in the cases where 10 or 20 iterations are the maximum.

Future directions. The algorithm presented above seems to offer further improvements, some of them will be summarized briefly:

- Try the bounding box pre-phase described above.
- Use $w \cdot z$ for a measure of how much w should be updated.
- An improvement (hopefully) of the incremental method: Store also those points , which are closest to the separating plane. For convex objects, we can find two points which are closest to the sep. plane by just doing a local minimization. The new closest points should be the stored ones or some in the vicinity! Actually, "closest" means wz is minimum/maximum for $z \in P/Q$, resp. If these new two "closest" points are still on the "right" sides (i.e.,

$wz^* > 0$ for optimum $z^* \in P$, $wz^* < 0$ for optimum $z^* \in Q$), then P and Q are still linearly separable.

The verification that a certain point is indeed closest to the separating plane can be done quickly by observing the following: if all neighbor vertices are further away from the plane, then this point is the closest point (the polyhedron is assumed to be convex).

If a certain point is not the closest point, we can just move towards the one which is closer. Since the polyhedron is convex, this will eventually terminate with the closest point.

3.5 Closed Objects

For closed polyhedra, we can define an *interior* and an *exterior* of the object. Collision of two closed polyhedra can be defined as:

$$\begin{aligned} P, Q \quad &\mathit{collide} \quad :\Leftrightarrow \\ \exists p : p &\in P \wedge p \in Q \end{aligned}$$

If we find such a point p, we're done. The problem with this approach is to prove that there is not such a point for given P and Q.

This suggests another probabilistic approach: generate a certain amount of random points and check if one of them is in P as well as in Q. If we find one, return "collision", if we don't, return "no collision". In contrast to the separating planes algorithm (see Section 3.4.2), this algorithm is biased towards "no collision".

An advantage of this algorithm, compared with the separating planes algorithm, is that it can be applied to a much larger class of polyhedra.

In order to decrease the error probability, we don't want to generate random points for which we know trivially that they are not in $P \cap Q$; this is the case for points which are not in $\mathrm{bbox}(P) \cap \mathrm{bbox}(Q)$.

Since the test points are randomly chosen, we could also choose a few special points. If P and Q intersect, then it is very probable that a vertex of P is inside Q or vice versa. Thus, we could first test the vertices of P (or just some of them) for containment in Q and vice versa; then, if none of them is contained, we try some other random points. In order to reduce the probability of "worst-case constellations", one should not make a contiguous pass over all vertices, because they tend to be close to each other (i.e., if a vertex is not in Q, then the succeeding vertex probably isn't either); instead, vertices should be chosen in a randomly fashion (this is called *stochastic pre-conditioning*).

A short excursion on probabilistic algorithms. A quick tour through Monte-Carlo algorithms will set the algorithm outlined above into the context of probabilistic algorithms [Reischuk88, BB88, BDG90]. I will do that by example of polyhedra and points.

Monte-Carlo algorithms are the class of probabilistic algorithms which always terminate, but which are occasionally wrong. Other classes are Las-Vegas, Sherwood, and approximation algorithms. The algorithm developed in this section is a Monte-Carlo algorithm.

We need to identify the language, which is in this particular case all pairs of polyhedra (together with their transformation), i.e.,
$\mathcal{L} = \{(P, Q) \mid P, Q \text{ polyhedra in } \mathbb{R}^3, P \cap Q \neq \varnothing\} \subseteq \{(P, Q) \mid \text{ polyhedra}\} = \Sigma^2$.

Now we need to choose the set of *witnesses* $\mathcal{W}$ which tell us, whether a pair (P, Q) is in $\mathcal{L}$. Here, $\mathcal{W} = \{w \in \mathbb{R}^3\}$. A *witnessing system* is a set of witnesses plus a predicate π such that

$$\forall x \in \Sigma \; \forall w \in \mathcal{W}: \quad \pi(x, w) = \text{true} \;\Rightarrow\; x \in \mathcal{L}$$

We will call such a w a *witness*. However, there might be *false witnesses*, for which $\pi(x, w) = \text{false} \;\wedge\; x \in \mathcal{L}$!

With the algorithm considered here, false witnesses are those test points which are not in $P \cap Q$, even though $P \cap Q \neq \varnothing$. The predicate π is a point-in-polyhedron test (see Section 3.5.1).

Error estimation. The average error of a witness w is the probability that w is a false witness. A Monte-Carlo algorithm is called *p-correct* if we can establish a lower bound on the "credibility" of all witnesses, i.e., if

$$P\left[\text{arbitrary, random } w \in \mathcal{W} \text{ is witness for } x\right] \;\geq\; p > 0$$

Unfortunately, it is not clear to me whether there is such a lower bound for the algorithm above, i.e., whether there is a $q > 0$ such that

$$P\left[w \in \text{bbox}(p) \cap \text{bbox}(Q) \text{ is witness } \middle| P \cap Q \neq \varnothing\right] = \frac{\text{vol}(p \cap Q)}{\text{vol}(\text{bbox}(P) \cap \text{bbox}(Q))} \geq q$$

3.5.1 Point-in-Polyhedron Test

The heart of the algorithm is the test whether a query point p is inside a polyhedron P or not. The algorithm chosen here is the one-shot method again, because it works for non-convex polyhedra, too; also, it proved to be very fast, because most of the work can be done by simple coordinate comparisons (see Section 3.2.2).

In contrast to the 2D case, the 3D case bears some intricacies: it is not trivial at all to resolve the *singular cases* where the ray emanating from the query point p hits an edge, a vertex, or lies in the plane of a polygon (see Figure 3.17)! In the 2D case, these can be solved by choosing the right comparisons. In the 3D case, several authors have tackled the problem [CT87, Kalay82, Linhart90].

The case "ray hits edge" can be decided by the two normals of the incident faces: if the two dot products of the ray and the normals have the same sign, then the ray will enter or leave the polyhedron; if the signs are opposite to each other, then the ray will stay inside or outside.

Kalay tries to resolve the case where the ray hits a vertex in the following way [Kalay82, Section 5.3]: project the normals of all faces incident to the vertex onto the ray; if all of them point in the same direction, then the ray will change sides. However, I think this is wrong: Figure 3.17(b) shows a simple counter-example.

Linhart [Linhart90] offers a solution by summing certain angles which are obtained by projecting all incident faces onto a plane normal to the ray. Then, the sign of the sum of angles determines whether to increment the "inside/outside"-counter or not.

However, with our special application in mind, we don't need to bother about these singular cases: if one should occur, we would just dismiss that query point and try another one.

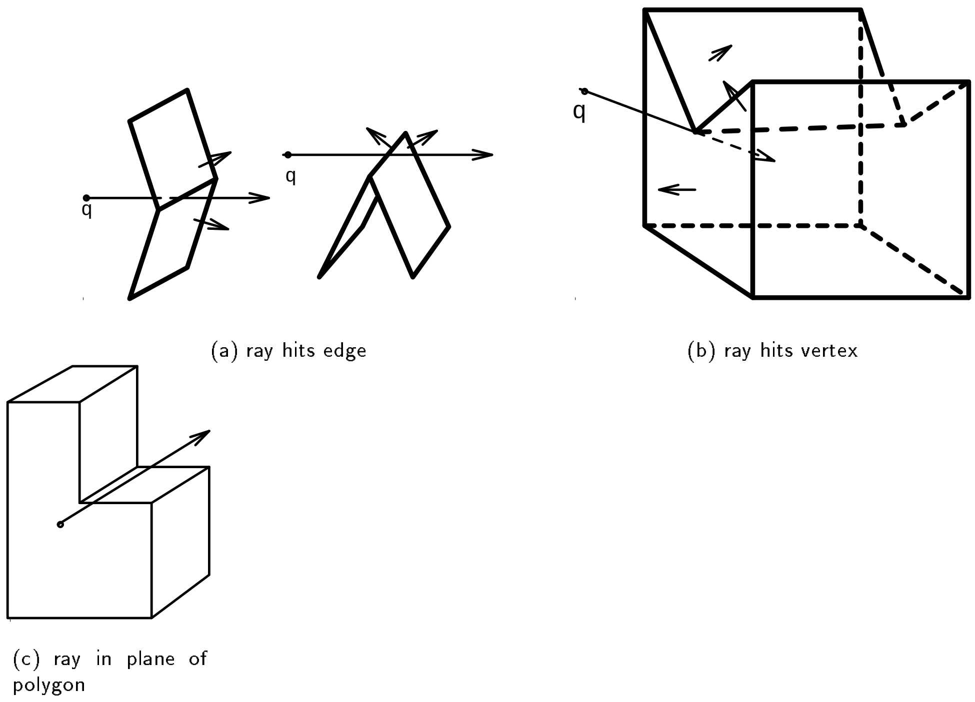


Figure 3.17. Singularities with point-in-polyhedron test

Results. Test runs (with 3 objects, 250 polygons in total, flying around in a cage of 6 boxes) showed the following results:

100 random points	17 frames/sec
test vertices first, then 100 random points	12 frames/sec
1000 random points	7 frames/sec

When all the vertices were tested first, it turned out that a collision was found by a random test point but not by a vertex in less than 1% of all collision detection requests; even with 1000 random test points.

3.6 Divide and Conquer

Although the exact algorithms presented in previous sections offer some possibilities for speed-up, profilings have shown that most of the time is spent in the inner loop over all polygons (which is inside the loop over all edges). 42% are spent in the single line which checks for an overlap of the edge bbox and another 40% are spent in the loop construct which loops over all faces!

So, the only way to speed up this sort of algorithms would have to be some method which allows us to skip many edges or faces at the same time, i.e., we have to eliminate the *pairwise weakness* (see Section 1.2.2, page 8). The idea is to use a *divide-&-conquer* approach. It was inspired by BSP trees, k-d trees, and *balanced bipartitions* (known in the area of VLSI layout algorithms; see [Lengauer90]). Of course, this involves a pre-processing step; and as such, this algorithm cannot be used whenever geometry changes during run-time.

The basic outline of the algorithm is (conceptually) as follows (see Figure 3.18): we divide the bounding boxes of P and Q into two halves (we call them "left" and "right" sub-box); we partition the set of edges of P into two sets depending whether

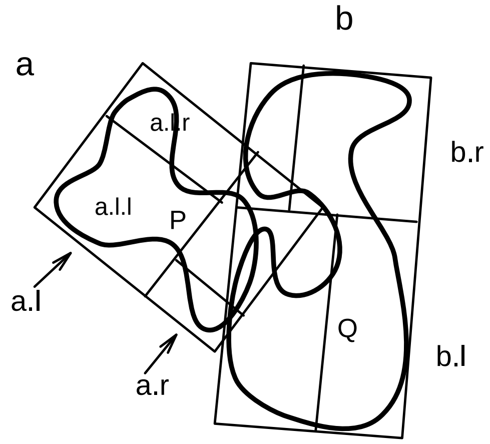


Figure 3.18. Only faces and edges of overlapping boxes have to be checked for intersection. For example, edges of a.l don't have to checked with polygons of b.l .

they are in the left or the right sub-box; in the same manner, we partition the set of polygons of Q. When checking edges of P and faces of Q for intersection, we first check whether bbox(P) intersects bbox(Q) (the non-aligned ones!); if they don't, we're finished. If they do, we check all 4 pairs of sub-boxes of P and Q, resp., for intersection. Obviously, we don't have to check edges against polygons, for which their boxes don't intersect.

Of course, the sub-box pre-processing is done recursively, which is why I will call the whole data structure a *box-tree*.

The intersection test of two boxes will be done by a variant of the Cyrus-Beck algorithm. For line clipping, there is a specialized version called *Liang-Barsky* algorithm [LB84]. (We assume that the reader be familiar with both algorithms.) We can do even better by exploiting the very special geometry of boxes and by clipping all box-edges parallel to each other at the same time; this will enable us to re-use many results during one step, and, even better, to re-use most of the intersection computations when descending down one level in the box-tree. Special features of boxes are:

- the faces form three sets of two parallel faces each,
- the edges form three sets of four parallel edges each,
- when a box is divided by a plane perpendicular to an edge, all edges retain their entering/leaving status.

3.6.1 Simultaneous Recursive Traversal of Box-Trees

The algorithm to be developed in this section has the following pseudo-code outline:

<u>`Simultaneous traversal of box-trees`</u>

`a` = *box in P's box-tree,* `b` = *box in Q's box-tree*
`a.l, a.r` *are left and right sub-boxes of* `a`

`a, b` *don't intersect* $\longrightarrow$ `exit`
`b` *leaf*
$\longrightarrow$
`a` *leaf*

```
        ⟶
            elementary operation on box-tree leaves
            return
        a not leaf
        ⟶
            a.l,b intersect   ⟶   traverse (a.l,b)
            a.r,b intersect   ⟶   traverse (a.r,b)
    b not leaf
    ⟶
        a leaf
        ⟶
            a,b.l intersect   ⟶   traverse (a,b.l)
            a,b.r intersect   ⟶   traverse (a,b.r)
        a not leaf
        ⟶
            a.l,b intersect
            ⟶
                a.l,b.l intersect   ⟶   traverse (a.l,b.l)
                a.l,b.r intersect   ⟶   traverse (a.l,b.r)
            a.r,b intersect
            ⟶
                a.r,b.l intersect   ⟶   traverse (a.r,b.l)
                a.r,b.r intersect   ⟶   traverse (a.r,b.r)
```

In this sub-section, we are not concerned with the "*elementary operation on box-tree leaves*". In fact, the simultaneous traversal of box-trees could be used for other operations, too: the only part that would have to be re-defined is that "*elementary operation*" – which provides the "semantics" of the overall operation (see [NAT90] for a similar point of view regarding BSP trees).

We are also not concerned with constructing the box-tree — this will be taken care of in Section 3.6.3.

Box-box intersection test. When we start descending two box-trees simultaneously, we initially have to check if bbox(P) and bbox(Q) intersect. This is done by clipping the edges of bbox(P) with bbox(Q) and vice versa (in fact, the method developed here is a special adaption of the algorithm of Section 3.4.1).

When clipping edges by a box, we will do this in the local coordinate system of the box (i.e., the box is axis-aligned with respect to this coordinate system).

We will consistently maintain two representations of each box: one in its own, local coordinate system (which is very simple), and one in the the local coordinate system of the other box.

Terms and notations. We have to define a few terms and notations. In the following, a box A will be represented by a point p, three unit vectors b^x, b^y, b^z, and $\Delta^P_x, \Delta^P_y, \Delta^P_z$ (see Figure 3.19). The point p will be called the *origin of the box* A; it will be maintained in the coordinate system of A and in the coordinate system of B.

The two planes which are perpendicular to the local x-axis b^x will be called *x-planes*; in particular, the x-plane which contains q will be called *xl-plane*; the x-plane which goes through $q + (\Delta^Q_x, 0, 0)$ will be called *xh-plane*. Similarly, we define y- and z-planes.

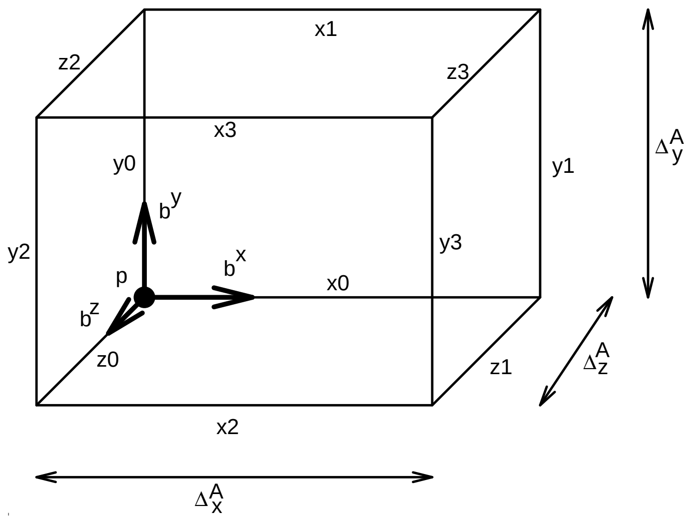


Figure 3.19. Designations of certain box features.

Edges are grouped into three families. Each family consists of the four edges of the box which are parallel to each other. They will be called x-, y-, and *z-edges*.

Given two boxes A and B, we will check if they intersect each other by calculating a parameter interval of every edge of A corresponding to that part of the edge which is inside B, and vice versa. Line parameters of edges of A will be denoted by t's, those of B by s's. The line interval of a clipped edge will be called *t-interval.* The line interval of edge $x1$ of A will be denoted by $T^{x1} = \left[T^{x1}_{\min}, T^{x1}_{\max}\right]$. Line parameters are always with respect to p or q, resp., i.e., $t_{\alpha 0} = 0$ is the point p for $\alpha \in \{x, y, z\}$.

t^{x0}_{xl} denotes that line parameter where edge $x0$ of A intersects with the xl-plane of B; all other t- and s-parameters are defined analogously.

First intersection test. This paragraph describes the procedure of the first intersection test of the roots of two boxtrees.

First, we compute the line intervals of the x-edges of A clipped at B (see Figure 3.20). To do that, we need to calculate the line intervals $t^{xi}_{..}$ of all edges intersected with all planes of B. Simple calculus yields $t^{x0}_{xl} = \frac{q_x - p_x}{b^x_x}$. Similarly,

$$t^{x0}_{xh} = \frac{q_x - p_x + \Delta^B_x}{b^x_x} = t^{x0}_{xl} + \frac{\Delta^B_x}{b^x_x}$$

which exploits the fact that the faces xl and xh are parallel. We can also use the fact that all x-edges are parallel:

$$t^{x1}_{xl} = \frac{(q - (p + \Delta^A_y b^y)) \cdot (-1, 0, 0)}{b^x \cdot (-1, 0, 0)} = t^{x0}_{xl} - \Delta^A_y \frac{b^y_x}{b^x_x}$$

Analogously, we can calculate all the other line parameters, which can be summarized in the following table:

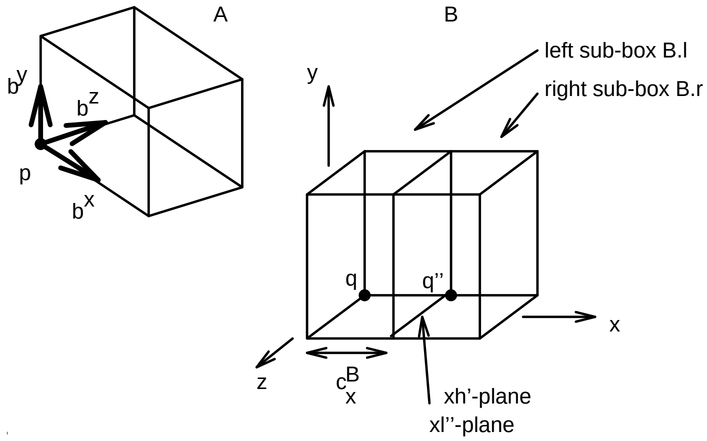


Figure 3.20. Splitting box B yields new line parameters for edges of A; cut-plane perpendicular to x-edges of B.

$t_{\cdot\cdot}^{\cdot\cdot\rightarrow}$ $\downarrow$	$x0$	$x1$	$x2$	$x3$
xl	$\frac{q_x - p_x}{b^x_x}$	$t^{x0}_{xl} - \Delta^A_y \frac{b^y_x}{b^x_x}$	$t^{x0}_{xl} - \Delta^A_z \frac{b^z_x}{b^x_x}$	$t^{x1}_{xl} - \Delta^A_z \frac{b^z_x}{b^x_x}$
xh	$t^{x0}_{xl} - \Delta^A_y \frac{b^y_x}{b^x_x}$	$t^{x0}_{xh} - \Delta^A_y \frac{b^y_x}{b^x_x}$	$t^{x0}_{xh} - \Delta^A_z \frac{b^z_x}{b^x_x}$	$t^{x1}_{xh} - \Delta^A_z \frac{b^z_x}{b^x_x}$
yl	$\frac{q_y - p_y}{b^x_y}$	$t^{x0}_{yl} - \Delta^A_y \frac{b^y_y}{b^x_y}$	$t^{x0}_{yl} - \Delta^A_z \frac{b^z_y}{b^x_y}$	$t^{x1}_{yl} - \Delta^A_z \frac{b^z_y}{b^x_y}$
yh	$t^{x0}_{yl} - \Delta^A_y \frac{b^y_y}{b^x_y}$	$t^{x0}_{yh} - \Delta^A_y \frac{b^y_y}{b^x_y}$	$t^{x0}_{yh} - \Delta^A_z \frac{b^z_y}{b^x_y}$	$t^{x1}_{yh} - \Delta^A_z \frac{b^z_y}{b^x_y}$
zl	$\frac{q_z - p_z}{b^x_z}$	$t^{x0}_{zl} - \Delta^A_y \frac{b^y_z}{b^x_z}$	$t^{x0}_{zl} - \Delta^A_z \frac{b^z_z}{b^x_z}$	$t^{x1}_{zl} - \Delta^A_z \frac{b^z_z}{b^x_z}$
zh	$t^{x0}_{zl} - \Delta^A_y \frac{b^y_z}{b^x_z}$	$t^{x0}_{zh} - \Delta^A_y \frac{b^y_z}{b^x_z}$	$t^{x0}_{zh} - \Delta^A_z \frac{b^z_z}{b^x_z}$	$t^{x1}_{zh} - \Delta^A_z \frac{b^z_z}{b^x_z}$

We will call this table a *t-table* for the x-edges of A. We need three t-tables for the three sets of x-, y-, and z-edges of A. There will be similar tables for the edges of B, which we will call *s-tables.*

Note that most of the terms in the table can be re-used; for example, the term $\Delta^A_z \frac{b^z_x}{b^x_x}$ has to be computed only once per table, i.e., we can get t^{x3}_{xl} and t^{x3}_{xh} by one multiplication and two additions.

Each quotient $\frac{b^i_j}{b^k_l}$ is needed exactly once for the calculation of all initial t- and s-tables; but we will need them later with the box-splitting step, so we will store them in a table. Most of the entries in a t-table can be computed by one multiplication and one addition.

In order to compute the t-intervals, we have to calculate the minimum and the maximum of the columns; there will be one minimum and one maximum per column, each on three values. The special geometry helps here, too:

$$\forall i : t^{xi}_{\cdot\cdot} \text{ } entering \;:\Leftrightarrow\; t^{x0}_{\cdot\cdot} \text{ } entering,$$
$$t^{\cdot\cdot}_{\cdot l} \text{ } entering \;:\Leftrightarrow\; t^{\cdot\cdot}_{\cdot h} \text{ } leaving.$$

This eliminates $\frac{3}{4}$ of all entering/leaving tests. Furthermore, the status of edges will be the same during the whole traversal down the box-tree; again, we can keep this information in an array of flags which has to be set up only once in the beginning of the traversal ($t^{x\cdot}_{\alpha l}$ *entering* $:\Leftrightarrow b^x_\alpha > 0$).

Descending the box-tree. For a given pair (a, b) of boxes, all the information on their intersection status is given by a set of $2 \times 3 \times 4$ line parameter intervals. The basic step of the traversal is the test "`a,b.l` *intersect*" and "`a,b.r` *intersect*". We will do this by bisecting the box b into its left and right sub-box, which is equivalent to computing two new sets of $2 \times 3 \times 4$ line parameter intervals, one describing $(a, b.l)$, the other describing $(a, b.r)$. In this sub-section we will describe the splitting of the box b; splitting a is quite analogous.

Splitting seems like a lot of computational work; however, half of the information stored in a set of line parameter intervals for (a, b) can be re-used for $(a, b.l)$, the other half for $(a, b.r)$.

New line parameters are always computed in the other box's local coordinate system. We will denote the new t-values (where a line intersects with a face) by $'t_{\cdot\cdot}^{\cdot\cdot}$ for edges of a clipped at $b.l$, and by $''t_{\cdot\cdot}^{\cdot\cdot}$ for edges of A clipped at $b.r$, resp.

Splitting B at c_x^B. All $'t$-values equal t-values except for $'t_{xh}^{\cdot\cdot}$ and $''t_{xl}^{\cdot\cdot}$ (see Figure 3.20). Fortunately, $'t_{xh}^{\cdot\cdot} = '' t_{xl}^{\cdot\cdot}$. So we have to compute the following values $'t_{xh}^{\alpha i}$:

$\alpha \backslash i$	0	1	2	3
x	$\frac{q_x'' - p_x}{b_x^x}$	$'t_{xh}^{x0} - \Delta_y^A \frac{b_x^y}{b_x^x}$	$'t_{xh}^{x0} - \Delta_z^A \frac{b_x^z}{b_x^x}$	$'t_{xh}^{x1} - \Delta_z^A \frac{b_x^z}{b_x^x}$
y	$\frac{q_x'' - p_x}{b_x^y}$	$'t_{xh}^{y0} - \Delta_x^A \frac{b_x^x}{b_x^y}$	$'t_{xh}^{y0} - \Delta_z^A \frac{b_x^z}{b_x^y}$	$'t_{xh}^{y1} - \Delta_z^A \frac{b_x^z}{b_x^y}$
z	$\frac{q_x'' - p_x}{b_x^z}$	$'t_{xh}^{z0} - \Delta_x^A \frac{b_x^x}{b_x^z}$	$'t_{xh}^{z0} - \Delta_y^A \frac{b_x^y}{b_x^z}$	$'t_{xh}^{z1} - \Delta_y^A \frac{b_x^y}{b_x^z}$

Again, $\frac{3}{4}$ of all entering/leaving tests can be eliminated by noticing that $'t_{xh}^{\cdot\cdot}$ *entering* $\Leftrightarrow$ $''t_{xl}^{\cdot\cdot}$ *leaving* $\Leftrightarrow$ $t_{xh}^{\cdot\cdot}$ *entering*.

These new $'t$- and $''t$-values are consistent with the old t-values in the sense that $'t = 0$ and $''t = 0$ describe the same point.

The new line parameter intervals of the clipped x-edges of A are:

$$\underrightarrow{t_{xh}^{\cdot\cdot} \text{ entering}} \qquad\qquad \{ \Leftrightarrow t_{xl}^{\cdot\cdot} \text{ leaving } \}$$

$$\begin{aligned} 'T^{\alpha i} &:= \left[\max\{T_{\min}^{\alpha i}, 't_{xh}^{\alpha i}\} \,,\, T_{\max}^{\alpha i} \right] \\ ''T^{\alpha i} &:= \left[T_{\min}^{\alpha i} \,,\, \min\{T_{\max}^{\alpha i}, 't_{xh}^{\alpha i}\} \right] \end{aligned}$$

$$\underrightarrow{t_{xh}^{\cdot\cdot} \text{ leaving}} \qquad\qquad \{ \Leftrightarrow t_{xl}^{\cdot\cdot} \text{ entering } \}$$

$$\begin{aligned} 'T^{\alpha i} &:= \left[T_{\min}^{\alpha i} \,,\, \min\{T_{\max}^{\alpha i}, 't_{xh}^{\alpha i}\} \right] \\ ''T^{\alpha i} &:= \left[\max\{T_{\min}^{\alpha i}, 't_{xh}^{\alpha i}\} \,,\, T_{\max}^{\alpha i} \right] \end{aligned}$$

The new origins of the left and right sub-boxes of B in B's own coordinate system are

$$\begin{aligned} q' &:= q \\ q'' &:= q + (c_x^B, 0, 0) \end{aligned}$$

In a similar manner, we have to split the s-tables of box B by using most of the values of the old s-values and computing the following new ones (see Figure 3.21).

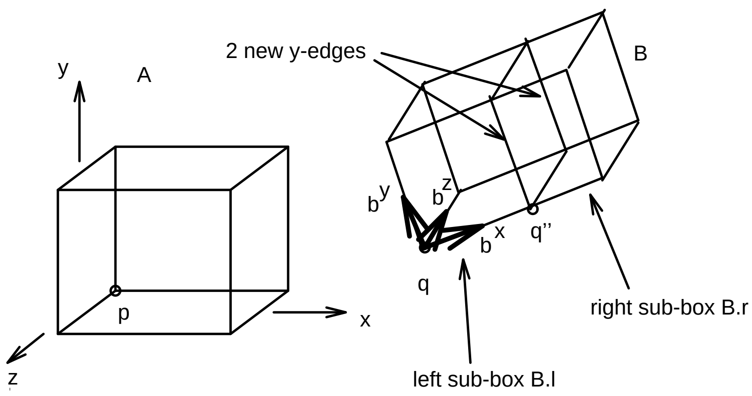


Figure 3.21. Splitting box B yields new line parameters for edges of B; cut-plane perpendicular to x-edges of B.

${}'s_{\cdot\cdot}^{\cdot\cdot}$	$y1$	$y3$	$z1$	$z3$
xl	$\frac{p_x - q''_x}{b^y_x}$	${}'s^{y1}_{xl} - \Delta^B_z \frac{b^z_x}{b^y_x}$	$\frac{p_x - q''_x}{b^z_x}$	${}'s^{z1}_{xl} - \Delta^B_y \frac{b^y_x}{b^z_x}$
xh	${}'s^{y1}_{xl} + \frac{\Delta^A_x}{b^y_x}$	${}'s^{y1}_{xh} - \Delta^B_z \frac{b^z_x}{b^y_x}$	${}'s^{z1}_{xl} + \frac{\Delta^A_x}{b^z_x}$	${}'s^{z1}_{xh} - \Delta^B_y \frac{b^y_x}{b^z_x}$
yl	$\frac{p_y - q''_y}{b^y_y}$	${}'s^{y1}_{yl} - \Delta^B_z \frac{b^z_y}{b^y_y}$	$\frac{p_y - q''_y}{b^z_y}$	${}'s^{z1}_{yl} - \Delta^B_y \frac{b^y_y}{b^z_y}$
yh	${}'s^{y1}_{yl} + \frac{\Delta^A_y}{b^y_y}$	${}'s^{y1}_{yh} - \Delta^B_z \frac{b^z_y}{b^y_y}$	${}'s^{z1}_{yl} + \frac{\Delta^A_y}{b^z_y}$	${}'s^{z1}_{yh} - \Delta^B_y \frac{b^y_y}{b^z_y}$
zl	$\frac{p_z - q''_z}{b^y_z}$	${}'s^{y1}_{zl} - \Delta^B_z \frac{b^z_z}{b^y_z}$	$\frac{p_z - q''_z}{b^z_z}$	${}'s^{z1}_{zl} - \Delta^B_y \frac{b^y_z}{b^z_z}$
zh	${}'s^{y1}_{zl} + \frac{\Delta^A_z}{b^y_z}$	${}'s^{y1}_{zh} - \Delta^B_z \frac{b^z_z}{b^y_z}$	${}'s^{z1}_{zl} + \frac{\Delta^A_z}{b^z_z}$	${}'s^{z1}_{zh} - \Delta^B_y \frac{b^y_z}{b^z_z}$

One can see, that only half of the terms $\Delta^B_\cdot \frac{b_\cdot^\cdot}{b_\cdot^\cdot}$ in the second and fourth column have to be computed.

The new ${}'S$- and ${}''S$-intervals are obtained from the old S-intervals by computing

$$
\begin{array}{ll}
{}'S^{y1} = {}''S^{y0} & {}'S^{y3} = {}''S^{y2} \\
{}'S^{z1} = {}''S^{z0} & {}'S^{z3} = {}''S^{z2}
\end{array}
$$

from scratch from the ${}'s$-values above; by shifting some S-intervals,

$$
\begin{aligned}
{}''S^{xi} &:= \left[\max\{0, S^{xi}_{\min} - c^B_x\}, S^{xi}_{\max} - c^B_x\right] \\
{}'S^{xi} &:= \left[S^{xi}_{\min}, \min\{S^{xi}_{\max}, c^B_x\}\right]
\end{aligned}
$$

and by copying some:

$$
\begin{array}{llll}
{}'S^{y0} := S^{y0} & {}'S^{y2} := S^{y2} & {}'S^{z0} := S^{z0} & {}'S^{z2} := S^{z2} \\
{}''S^{y1} := S^{y1} & {}''S^{y3} := S^{y3} & {}''S^{z1} := S^{z1} & {}''S^{z3} := S^{z3}
\end{array}
$$

That is, 4 intervals (out of 12) have really to be computed.

The new origins of the left and right sub-boxes of B in A's local coordinate system are

$$
\begin{aligned}
q' &:= q \\
q'' &:= q + c^B_x b^x
\end{aligned}
$$

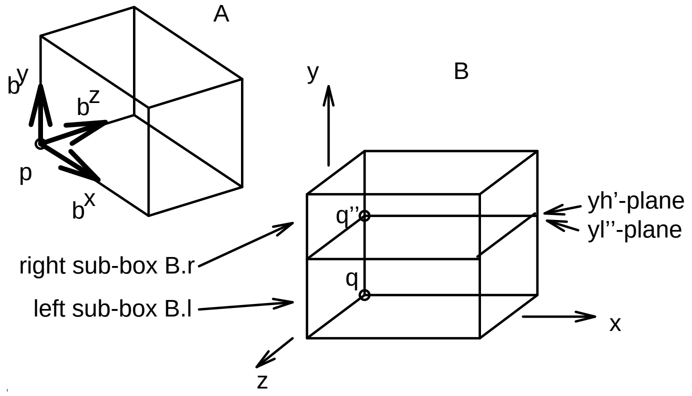


Figure 3.22. Splitting box B yields new line parameters for edges of A; cut-plane perpendicular to y-edges of B.

Splitting B at c_y^B. This is quite similar to splitting it at c_x^B; we include calculations here just for sake of completeness. We have to compute values $'t_{yh}^{\alpha i}$ (see Figure 3.22):

$\alpha\backslash i$	0	1	2	3
x	$\frac{q''_y - p_y}{b_y^x}$	$'t_{yh}^{x0} - \Delta_y^A \frac{b_y^y}{b_y^x}$	$'t_{yh}^{x0} - \Delta_z^A \frac{b_y^z}{b_y^x}$	$'t_{yh}^{x1} - \Delta_z^A \frac{b_y^z}{b_y^x}$
y	$\frac{q''_y - p_y}{b_y^y}$	$'t_{yh}^{y0} - \Delta_x^A \frac{b_y^x}{b_y^y}$	$'t_{yh}^{y0} - \Delta_z^A \frac{b_y^z}{b_y^y}$	$'t_{yh}^{y1} - \Delta_z^A \frac{b_y^z}{b_y^y}$
z	$\frac{q''_y - p_y}{b_y^z}$	$'t_{yh}^{z0} - \Delta_x^A \frac{b_y^x}{b_y^z}$	$'t_{yh}^{z0} - \Delta_y^A \frac{b_y^y}{b_y^z}$	$'t_{yh}^{z1} - \Delta_y^A \frac{b_y^y}{b_y^z}$

The new $'T$- and $''T$-intervals are obtained like above.

The new $'s$-values to be computed are (see Figure 3.23):

$'s_{\cdot\cdot}^{\cdot\cdot}$	$x1$	$x3$	$z2$	$z3$
xl	$\frac{p_x - q''_x}{b_x^x}$	$'s_{xl}^{x1} - \Delta_z^B \frac{b_x^z}{b_x^x}$	$\frac{p_x - q''_x}{b_x^z}$	$'s_{xl}^{z2} - \Delta_x^B \frac{b_x^x}{b_x^z}$
xh	$'s_{xl}^{x1} + \frac{\Delta_x^A}{b_x^x}$	$'s_{xh}^{x1} - \Delta_z^B \frac{b_x^z}{b_x^x}$	$'s_{xl}^{z2} + \frac{\Delta_x^A}{b_x^z}$	$'s_{xh}^{z2} - \Delta_x^B \frac{b_x^x}{b_x^z}$
yl	$\frac{p_y - q''_y}{b_y^x}$	$'s_{yl}^{x1} - \Delta_z^B \frac{b_y^z}{b_y^x}$	$\frac{p_y - q''_y}{b_y^z}$	$'s_{yl}^{z2} - \Delta_x^B \frac{b_y^x}{b_y^z}$
yh	$'s_{yl}^{x1} + \frac{\Delta_y^A}{b_y^x}$	$'s_{yh}^{x1} - \Delta_z^B \frac{b_y^z}{b_y^x}$	$'s_{yl}^{z2} + \frac{\Delta_y^A}{b_y^z}$	$'s_{yh}^{z2} - \Delta_x^B \frac{b_y^x}{b_y^z}$
zl	$\frac{p_z - q''_z}{b_z^x}$	$'s_{zl}^{x1} - \Delta_z^B \frac{b_z^z}{b_z^x}$	$\frac{p_z - q''_z}{b_z^z}$	$'s_{zl}^{z2} - \Delta_x^B \frac{b_z^x}{b_z^z}$
zh	$'s_{zl}^{x1} + \frac{\Delta_z^A}{b_z^x}$	$'s_{zh}^{x1} - \Delta_z^B \frac{b_z^z}{b_z^x}$	$'s_{zl}^{z2} + \frac{\Delta_z^A}{b_z^z}$	$'s_{zh}^{z2} - \Delta_x^B \frac{b_z^x}{b_z^z}$

From this table we compute

$$
\begin{array}{ll}
'S^{x1} = ''S^{x0} & 'S^{x3} = ''S^{x2} \\
'S^{z2} = ''S^{z0} & 'S^{z3} = ''S^{z1}
\end{array}
$$

We copy

$$
\begin{array}{lcl}
S^{x0}, S^{x2}, S^{z0}, S^{z1} & \Longrightarrow & 'S \\
S^{x1}, S^{x3}, S^{z1}, S^{z3} & \Longrightarrow & ''S
\end{array}
$$

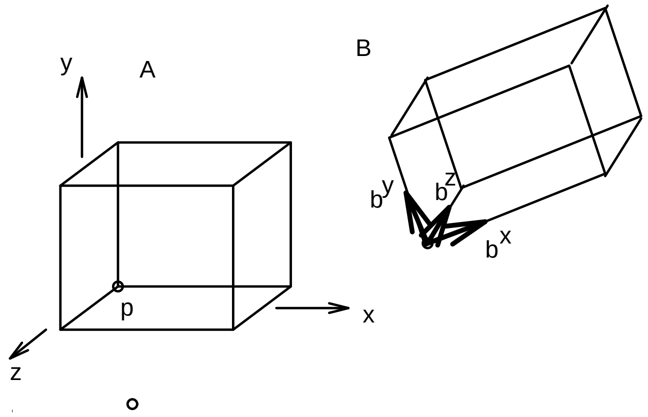


Figure 3.23. Splitting box B yields new line parameters for edges of B; cut-plane perpendicular to y-edges of B.

We shift/clip

$$\begin{aligned} {}''S^{yi} &:= \left[\max\{0, S^{yi}_{\min} - c^B_y\}, S^{yi}_{\max} - c^B_y\right] \\ {}'S^{yi} &:= \left[S^{yi}_{\min}, \min\{S^{yi}_{\max}, c^B_y\}\right] \end{aligned}$$

The new origins of the left and right sub-boxes of B in B's own coordinate system are

$$\begin{aligned} q' &:= q \\ q'' &:= q + (0, c^B_y, 0) \end{aligned}$$

and in A's local coordinate system they are

$$\begin{aligned} q'' &:= q + c^B_y b^y \\ q' &:= q \end{aligned}$$

Splitting B at c^B_z. We have to compute values ${}'t^{\alpha i}_{zh}$ (see Figure 3.24):

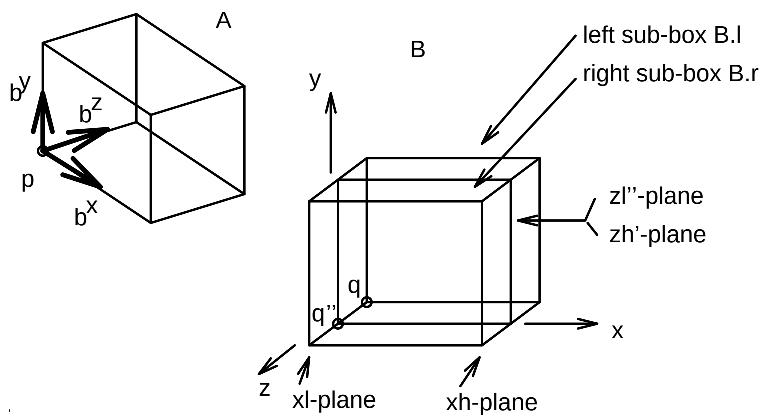


Figure 3.24. Splitting box B yields new line parameters for edges of A; cut-plane perpendicular to z-edges of B.

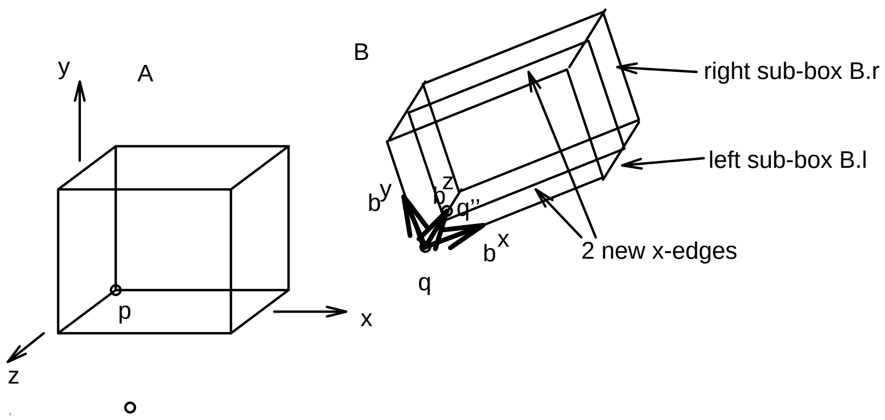


Figure 3.25. Splitting box B yields new line parameters for edges of B; cut-plane perpendicular to z-edges of B.

$\alpha \backslash i$	0	1	2	3
x	$\frac{q''_z - p_z}{b^x_z}$	$'t^{x0}_{zh} - \Delta^A_y \frac{b^y_z}{b^x_z}$	$'t^{x0}_{zh} - \Delta^A_x \frac{b^z_z}{b^x_z}$	$'t^{x1}_{zh} - \Delta^A_x \frac{b^z_z}{b^x_z}$
y	$\frac{q''_z - p_z}{b^y_z}$	$'t^{y0}_{zh} - \Delta^A_z \frac{b^x_z}{b^y_z}$	$'t^{y0}_{zh} - \Delta^A_z \frac{b^z_z}{b^y_z}$	$'t^{y1}_{zh} - \Delta^A_y \frac{b^z_z}{b^y_z}$
z	$\frac{q''_z - p_z}{b^z_z}$	$'t^{z0}_{zh} - \Delta^A_z \frac{b^x_z}{b^z_z}$	$'t^{z0}_{zh} - \Delta^A_z \frac{b^y_z}{b^z_z}$	$'t^{z1}_{zh} - \Delta^A_y \frac{b^y_z}{b^z_z}$

See Figure 3.25:

$'s^{\cdot\cdot\rightarrow}_{\cdot\cdot\downarrow}$	$x2$	$x3$	$y2$	$y3$
xl	$\frac{p_x - q''_x}{b^x_x}$	$'s^{x2}_{xl} - \Delta^B_y \frac{b^y_x}{b^x_x}$	$\frac{p_x - q''_x}{b^y_x}$	$'s^{y2}_{xl} - \Delta^B_x \frac{b^x_x}{b^y_x}$
xh	$'s^{x2}_{xl} + \frac{\Delta^A_x}{b^x_x}$	$'s^{x2}_{xh} - \Delta^B_y \frac{b^y_x}{b^x_x}$	$'s^{y2}_{xl} + \frac{\Delta^A_x}{b^y_x}$	$'s^{y2}_{xh} - \Delta^B_x \frac{b^x_x}{b^y_x}$
yl	$\frac{p_y - q''_y}{b^x_y}$	$'s^{x2}_{yl} - \Delta^B_y \frac{b^y_y}{b^x_y}$	$\frac{p_y - q''_y}{b^y_y}$	$'s^{y2}_{yl} - \Delta^B_x \frac{b^x_y}{b^y_y}$
yh	$'s^{x2}_{yl} + \frac{\Delta^A_y}{b^x_y}$	$'s^{x2}_{yh} - \Delta^B_y \frac{b^y_y}{b^x_y}$	$'s^{y2}_{yl} + \frac{\Delta^A_y}{b^y_y}$	$'s^{y2}_{yh} - \Delta^B_x \frac{b^x_y}{b^y_y}$
zl	$\frac{p_z - q''_z}{b^x_z}$	$'s^{x2}_{zl} - \Delta^B_y \frac{b^y_z}{b^x_z}$	$\frac{p_z - q''_z}{b^y_z}$	$'s^{y2}_{zl} - \Delta^B_x \frac{b^x_z}{b^y_z}$
zh	$'s^{x2}_{zl} + \frac{\Delta^A_z}{b^x_z}$	$'s^{x2}_{zh} - \Delta^B_y \frac{b^y_z}{b^x_z}$	$'s^{y2}_{zl} + \frac{\Delta^A_z}{b^y_z}$	$'s^{y2}_{zh} - \Delta^B_x \frac{b^x_z}{b^y_z}$

$'S^{x2} = ''S^{x0} \quad 'S^{x3} = ''S^{x1}$
$'S^{y2} = ''S^{y0} \quad 'S^{y3} = ''S^{y1}$

$$\begin{aligned} S^{x0}, S^{y0}, S^{x1}, S^{y1} &\implies 'S \\ S^{x2}, S^{x3}, S^{y2}, S^{y3} &\implies ''S \end{aligned}$$

$'S^{zi}$ and $''S^{zi}$ are shifted/clipped like above.

3.6.2 Parallelism

This is an ubiquitous, annoying issue, with which we have to deal, too. Edges won't be parallel to faces too often; but it can happen in starting positions, where objects tend to be aligned to each other and to the world coordinate system. (It does happen at start-up of test programs ...)

Like with many other properties or values, parallelism is preserved during simultaneous traversal through the box-tree. Again, the special geometry of boxes reduces the number of significant edge-face comparisons: if an edge is parallel to a face, then all edges of the same family are parallel to all faces of the other family. We can set up an array of flags in the beginning, which describes these relations.

While bisecting a box, we have to compute four new T- or S-intervals. During that computation, we might discover that an edge is parallel to a plane, at which we were about to clip it. Two things could happen: either, the edge is on the "wrong" side of the plane, in which case the interval will be empty and we're finished; or, the edge is on the "right" side, in which case we just proceed to the next plane.

The procedure how to handle the "parallel" case will be illustrated with two examples. First, we'll consider the case where the x-edges of A are parallel to the z-planes (zl and zh) of B (i.e., $b^x_z = 0$). This implies that the $T^{x\cdot}$-intervals will be treated specially when splitting B across its z-edges: they will be left alone or eliminated.

$$\begin{aligned}
p_z \geq q''_z &\quad\rightarrow\quad {}'T^{x0} := \varnothing\,,\ {}''T^{x0} := T^{x0} \\
p_z + \Delta^A_y b^y_z \geq q''_z &\quad\rightarrow\quad {}'T^{x1} := \varnothing\,,\ {}''T^{x1} := T^{x1} \\
p_z + \Delta^A_z b^z_z \geq q''_z &\quad\rightarrow\quad {}'T^{x2} := \varnothing\,,\ {}''T^{x2} := T^{x2} \\
p_z + \Delta^A_y b^y_z + \Delta^A_z b^z_z \geq q''_z &\quad\rightarrow\quad {}'T^{x3} := \varnothing\,,\ {}''T^{x3} := T^{x3}
\end{aligned}$$

$$\begin{aligned}
&p_z < q''_z \quad\rightarrow\quad {}'T^{x0} := T^{x0}\,,\ {}''T^{x0} := \varnothing \\
&etc. \ldots
\end{aligned}$$

The second example considers the y-edges of B being parallel to the x-planes of A. Let $d = p - q$, then

$$\begin{aligned}
&-d_x < 0 \lor -d_x > \Delta^A_x \\
&\longrightarrow \quad {}'S^{y1} := {}''S^{y0} := \varnothing \\
&(-d + \Delta^B_z \cdot b^z)_x < 0 \ \lor\ (-d + \Delta^B_z \cdot b^z)_x > \Delta^A_x \\
&\longrightarrow \quad {}'S^{y3} := {}''S^{y2} := \varnothing
\end{aligned}$$

3.6.3 Constructing the Box-Tree

The box-trees being constructed here are inspired by *k-d trees* (see [BF79]) and *balanced bipartitions* from VLSI layout algorithms (see [Lengauer90]). We do not construct octrees because they seem to be too inflexible: for reasons which will be clear in a moment, we don't want to subdivide a box by three planes at the same time and only in the middle (in fact, octrees are a special form of k-d trees).

The discussion will be restricted to the construction of boxtrees for a set polygons. Everything carries over to boxtrees for edges quite analogously.

The goal is to partition recursively the set of polygons (and edges, resp.) in such a way that the intersection test between two such partitions involves a minimum of elementary (i.e., edge-polygon) intersection tests. In the following, we will try to derive some heuristics for an optimal partitioning.

Whenever the collision detection algorithm goes down one level, and it discards one of the sub-boxes, we want as much polygons as possible to be discarded. This leads to the heuristic that in each of the two sub-boxes of a box there should be the same number of polygons. In general, there will be always polygons which are

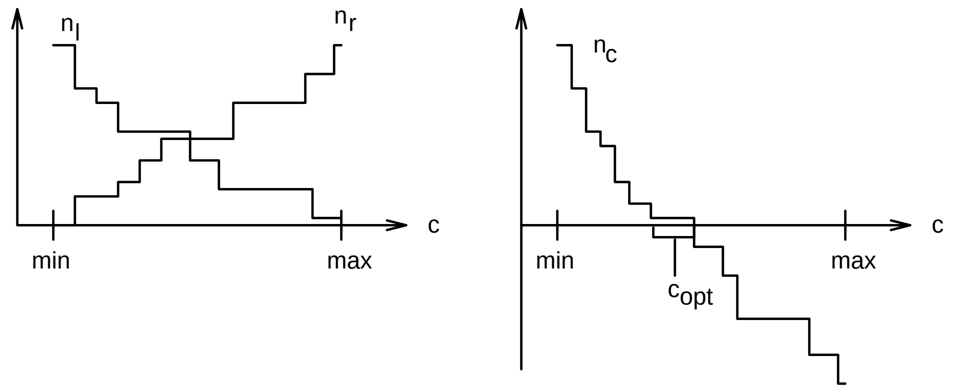


Figure 3.26. The penalty function is monotonic when not taking crossing polygons into account. The graphs of n_l and n_r are not necessarily mirror images of each other, since crossing polygons are not accounted for.

in both sub-boxes, though. When going down one level in the box-tree, we have to deal with those twice, once in the left-sub-box and once in the right sub-box. This leads to the heuristic that a bisection of a box should cut as few polygons as possible.

We start with a set of n polygons which are all contained in a box of size $[(x,y,z)_{\min},(x,y,z)_{\max}]$. Given a cut-plane c through that box, and perpendicular to the x-axis (w.l.o.g.), we denote the number of polygons to the left, the right, and crossing c by n_l, n_r, and n_c, resp. According to the heuristic proposed above, we define a *penalty function* for c by

$$p(c) = |n_l - n_r| + \gamma n_c$$

where γ is the factor by which a crossing polygon is worse than an unbalanced one. (Note: in general, $n_l + n_r + n_c \geq n$.)

The basic step for building a boxtree is to find the cutplane c for a given set of polygons such that c realizes the *global minimum*

$$\min \begin{Bmatrix} \min\{p(c) \mid c \perp \mathrm{x}-\mathit{axis}, c_x \in [x_{\min}, x_{\max}]\} \\ \min\{p(c) \mid c \perp y-\mathit{axis}, c_y \in [y_{\min}, y_{\max}]\} \\ \min\{p(c) \mid c \perp z-\mathit{axis}, c_z \in [z_{\min}, z_{\max}]\} \end{Bmatrix}$$

If we had chosen the penalty function to be $p(c) = |n_l - n_r|$ without taking crossing polygons into account, then finding the minimum would be easy, because $p(c)$ would be monotonic (see Figure 3.26). In that case we could simply use *interval bisection.*

We still use that method, and results have been satisfactory, i.e., we use $p'(c) = |n_l - n_r|$ to calculate a cut-plane along one axis; for choosing the final cut-plane from one of the three, we use $p(c)$.

To calculate a (local) minimum for, say, the x-axis, we start with $l := x_{\min}$, $r := x_{\max}$, and $c := l + r$.

$$\begin{aligned} n_l < n_r \quad &\rightarrow \quad l' := \frac{n_l}{n_l+n_r} l + \frac{n_r - n_l}{n_l + n_r} c \\ n_l \geq n_r \quad &\rightarrow \quad r' := \frac{n_r}{n_l+n_r} r + \frac{n_l - n_r}{n_l + n_r} c \end{aligned}$$

This is a little modification of the simple interval bisection; we try to take into account how far away we are from the optimum, so as to minimize the number of iterations. The iteration stops, if

- $p(c) \leq p_{\max}$. (typ. $p_{\max} \approx 0.5$), or
- # iterations > Max. (typ. Max ≈ 10), or
- $|l - r| <$ Min. (typ. Min $\approx$ the machine resolution)

When we have found a cut-plane, we divide the input array of polygons into two; the crossing polygons are copied into both (for reasons which will be made clear below). Then we start the process over for the two new arrays.

The box splitting recursion will stop when one of the following conditions holds:

- depth $\geq d_{\max}$.
- # polygons in the box currently considered for splitting $\leq$ Min.
- one of the two sub-boxes wouldn't contain any polygons.
- $n_l > \lambda n$ or $n_r \lambda n$ (it doesn't make sense to split the box, if one of the sub-boxes contains almost as much polygons as the father).

When the recursion stops, we attach the array of polygons to the corresponding leaf of the box-tree.

Crossing polygons. What should we do with crossing polygons (polygons which are on both sides of the cut plane)? If we stored them at inner nodes of the box-tree, there would be two possibilities to deal with them during the traversal for collision detection:

1. We check those against all the edges which are at or beneath the other box. This can become inefficient, because crossing polygons could be discarded anyway on one of the next levels of the simultaneous traversal (see Figure 3.27).
2. Otherwise, we have to provide another elementary operation "*check polygon with a box-tree*". This could become very expensive, since we have to do the initial computations for every polygon! (see above, "First intersection test")

The approach we have taken is to store polygons only at leaves. So, crossing polygons will be put in both sub-boxes. This avoids the disadvantages mentioned above. Of course, polygons can be stored multiple times at leaves, this way. However, this does not cause any memory problems: tests have shown that a box-tree contains by about a factor of 1.2 . . . 1.6 more pointers to edges/faces than there really are.

Geometrical robustness. Although this seems to be of minor importance, experiments showed its necessity very early. This is especially true for objects which are computer-generated and expose a very regular symmetry, like spheres, tori, regularly sub-divided quadrangles, extruded and revolved objects, etc. These objects usually have very good cut-planes, but if the splitting algorithm is not robust, the box-tree will be totally useless.

In general, geometrical robustness is an issue for all kinds of geometrically subdividing or splitting algorithms (see [BD92a] for another example).

Here the problem is: when do we consider a polygon to be on the left, the right, or crossing the cut-plane? Because of numerical inconsistencies, many polygons might be classified "crossing" even though they only touch the cut-plane (see Figure 3.28). The idea is simply to give the cut-plane a certain "thickness" 2δ. Then, we'll still consider a polygon left of a cut-plane c, even if one of its edges is right of c, but left of $c + \delta$. All the possible cases are depicted in Figure 3.28.

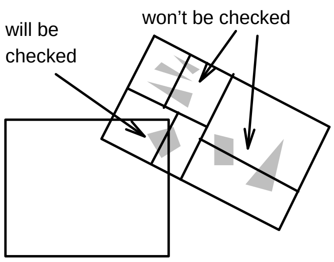


Figure 3.27. If crossing polygons are stored with the leaves of the box-tree, too, they can be discarded during the simultaneous traversal like "non-crossing" polygons.

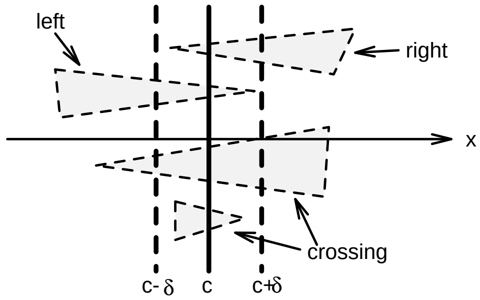


Figure 3.28. For splitting a set of polygons by a plane, geometrical robustness can be achieved by giving the cut-plane a certain "thickness".

Complexity. The complexity of constructing box-trees will be shown to be in $O(n)$, where n is the number of polygons, under certain assumptions. Quite similarly, the memory complexity can be shown to be in $O(n)$. Both results have been confirmed experimentally.

We assume that cutting a box takes s passes over all polygons in that box; we assume further that every cut will split the box's polygons into 3 sets of equal size: left, right, and crossing polygons. This means that a sub-box gets $\frac{2}{3}$ of the box's polygons. Thus, the depth of a box-tree will be $d = \log_{\frac{3}{2}}(n)$.

Let $T(n)$ be the time needed to build a box-tree for n polygons. Then,

$$\begin{aligned}
T(3k) &= cn + 2T(k), \\
T(3) &= c, \\
T(3^d) &= c3^d + 2\left(c3^{d-1} + 2\left(c3^{d-2}+\right)\right) \\
&= c3^d\left(1 + \frac{2}{3} + \frac{4}{3^2} + \dots\right) \\
&\leq c3^d 2\sum_{0}^{d-1}\frac{2^i}{3^i} \\
&\leq 3^{d+1}c \\
&\in O(n)
\end{aligned}$$

Other criteria for box-splitting. It is not clear, yet, which heuristic for building box-trees would be the best. If we allowed *empty leaves* in the tree, then we might be able to quit the collision detection traversal very early, because there is no collision with empty boxes. It is *not* necessarily best to try to keep the boxes as cubic as possible (which could be achieved by just splitting boxes along the longest of their edges). Altogether, a new heuristic for the cut-plane could be:

1. try to find a cut-plane which is near the middle of the edge (say, within a third) so that there are no polygons in one of the sub-boxes;
2. if there isn't such a plane, use the old heuristic;
3. if all three potential cut-planes (see Equation 3.6.3 (along x-, y-, and z-axis) have similar penalty values, choose the one which cuts the longest edge.

We could implement the first part of the heuristic by trying to push a plane from the left towards the middle of the edge until the left sub-box would contain a polygon.

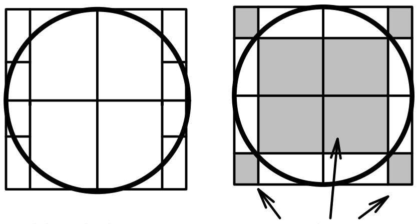


Figure 3.29. It might be better to allow empty boxes in a box-tree, too.

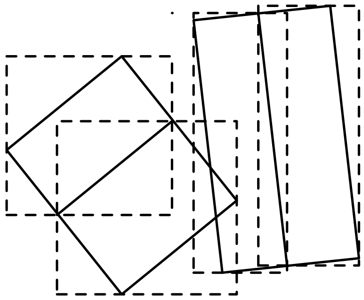


Figure 3.30. The simultaneous traversal of a box-tree might be faster when computing axis-aligned boxes on-the-fly and checking those instead of the original ones.

We would do likewise from the right, and choose that plane which got closer to the middle of the edge. Of course, this had to be done for all three axes.

3.6.4 Conclusion

Future work. The algorithm presented above seems to offer many more possibilities for further speed-up:

- Try a simultaneous traversal with *axis-aligned boxes* (see Figure 3.30). They can be computed on-the-fly from the ones on the level above together with the information stored with each box-tree node (i.e., the direction of the cut-plane and the cut-coordinate). Still, build the box-tree like we did so far.
- *Allow empty boxes.* Try *other criteria* for the computation of the "optimal" cut-plane with box-trees: boxes should be as close to cubes as possible; large empty boxes should be rewarded, too.
- Would it help, if we allowed boxes only to be *split in the middle*? I.e., with every recursion, one of the Δ's would be divided by 2. So, one could set up the $\frac{b^i_j}{b^k_l}$-tables in the beginning of the collision-check function, and also compute the $\Delta_{x,y,z}\frac{b^i_j}{b^k_l}$-tables for all recursion levels.

Implementation. The "*elementary operation*" (see Section 3.6.1), which operates on two leaves of the box-tree, is the usual algorithm presented in Section 3.2 (in fact, much of the code developed there is re-used). Of course, the same bounding box pre-checks are done here, too. The function doing the recursive simultaneous traversal through the box-tree can be written quite generic, so as to allow for different "semantics"; whenever it hits the bottom, i.e., it is called with two leaves, it will execute a callback, which provides the semantics.

For the time being, a complete boxtree for a polyhedron is constructed by first building one for polygons. In a second phase, the edges are inserted in the tree. As with the algorithms of Sections 3.2 and 3.4.1, there are two phases: check edges of polyhedron P against faces of Q, and vice versa; this is easily accomplished by calling the same function for simultaneous box-tree traversal twice (see next paragraph for a little comment on that).

The box-trees are generated at "announcement time" to the collision detection module.

The initial t- and s-tables can be computed by the same functions which do the box-splitting, we just call them once with $q'' = q$ and $p'' = p$, and once with $q'' = q + \Delta^B_\alpha$ and $p'' = p + \Delta^A_\alpha$, resp.

The t- and s-values itself are only temporary; they don't have to be kept after the new $'T$- and $''T$-intervals have been derived.

If a box-edge is completely outside the box, then it doesn't have to be considered any more with further box bisections. We can save a little work during the traversal by keeping an "outside" flag for every edge, and by decrementing a counter (which is initialized to 12) with every edge which is found to be outside. Then we can decide very quickly if two boxes do not intersect.

When computing $'T^{\alpha i}$ and $''T^{\alpha i}$, one should use the fact that $T^{\alpha i}_{\min} \leq T^{\alpha i}_{\max}$ always. This saves 0.5 comparisons on average.

Of course, the function building the box-trees should be implemented in a generic way. Actually, this is very easy: it suffices to write a function, which takes an array of boxes and splits those using the rules from above. Then, in order to build the polygon box-tree and the edge box-tree, the only difference is the "front-end" which sets up the initial array of boxes enclosing polygons or edges, resp.

Each node in the box-tree stores only:

- the name of the (local) axis which the box has to be divided along in order to get its sub-boxes,
- the position where the plane cuts the axis,
- two pointers to the left and right sub-box,
- for leaves only: a pointer to the data which is attached to the leaf.

A test program has been implemented, too; it was great help with finding bugs in the implementation. Also, it can be of great help in finding optimal box-splitting heuristics. An example of the collision detection algorithm at work can be found in Figure 3.31.

3.6.5 Results

The very first result was obtained on SGI's Skywriter (VGX, 8× R3000, 40 MHz): the box-tree algorithm was about 2× as slow as the general algorithm (Section 3.2) for arbitrary polyhedra! (Which was pretty shocking, `:-)`) However, on SGI's Onyx and Indigo, the box-tree algorithm is indeed much faster then the general one.

The maintaining of flags for empty intervals, which afterwards never have to be touched again, gives a speed-up of about 14% in the worst-case[4]. It does not help in the best-case, because the best-case does not traverse the box-tree at all.

Another speed-up was gained by implementing the collect phase within the elementary operation *leaf-box vs. leaf-box* (see Section 3.6.1), which throws away all polygons of a leaf-box that are not contained within the other object's bounding box. The speed-up gained by that phase is about a factor of 1.5.

[4] `movem -x 20 -c -a bx -b 10 -d 10`, resp., which is the zoo of 3 (5?) objects, altogether 15,000 polygons (no rendering, runtime 1h)

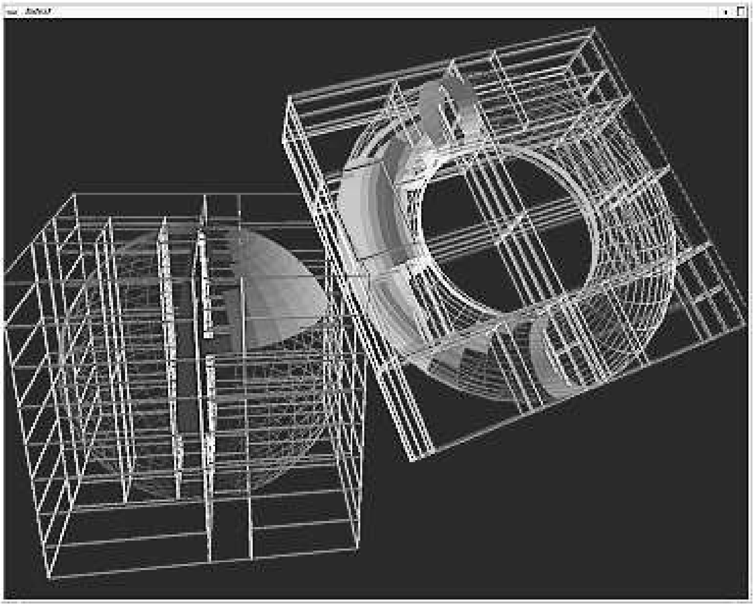


Figure 3.31. This visualization of the box-tree algorithm shows, how many and which polygons are actually considered for intersection. The leaves of the boxtree are depicted graphically by boxes. Boxes which intersect with any box of the other object's box-tree are highlighted.

Optimal box-tree parameters. I also tried to find out the optimum parameters for a box-tree, namely the maximum depth and the minimum number of polygons/edges within a box. To this end, I ran several tests with different objects and different choices of the parameters[5]. I varied only one parameter at a time. Each sample is the average over 2000 frames. All these tests were run on an Indigo with a 50 MHz R4000; rendering was switched off. Results are presented in Figures 3.32,3.33,3.34.

Comparison with conventional algorithms. Next, I compared the box-tree algorithm (using optimum parameters during box-tree construction) to the conventional algorithm. Timing was done for three different objects, a sphere (which is convex), a torus, and a tetra-flake (a very "non-convex" object). Again, two objects of the same kind were bouncing off each other in a cage. Each sample is the average over 10,000 frames. The tests were run on an Onyx (2× R4400 150MHz), rendering was switched off[6].

As expected, box-trees are much faster above a certain object complexity, but slower for small objects. The following table lists the thresholds above which the box-tree algorithm is faster, and the speed-up factor (all values are estimates from Figures 3.35, 3.36, and 3.37) $T_{\text{conventional}}(n)/T_{\text{box-tree}}(n)$ for $n = 50, 100, 500, 1000$ polygons per object:

object type	threshold	$T_{\text{conventional}}/T_{\text{box-tree}}$			
		n=50	n=100	n=500	n=1000
sphere	300	0.47	0.36	1.18	2.25
torus	60	0.9	1.58	4.6	200
tetra-flake	300	0.4	0.5	1.6	2.1

[5] movem -t 2000 <obj.type> -x <size> -d <maxdepth> -b <minperbox> -v bs -e 1.2 -a bx -s n 1

[6] Invocation: movem -x n -t 10000 to -s n 1 -e 1.0 -a bx -v bs

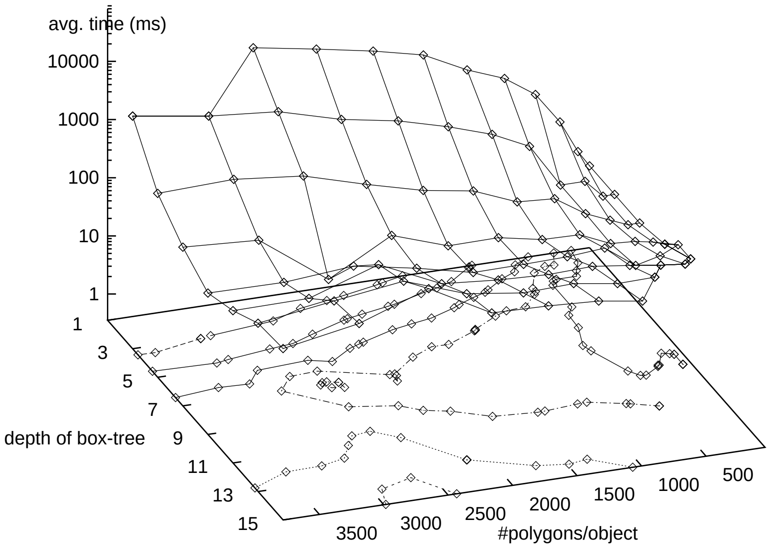


Figure 3.32. Search for the optimal maximum depth of a boxtree. Test objects are two spheres of varying complexity. The minimum number of elements per box was fixed to 1. Below a complexity of 50 polygons per object, no significant change could be measured.

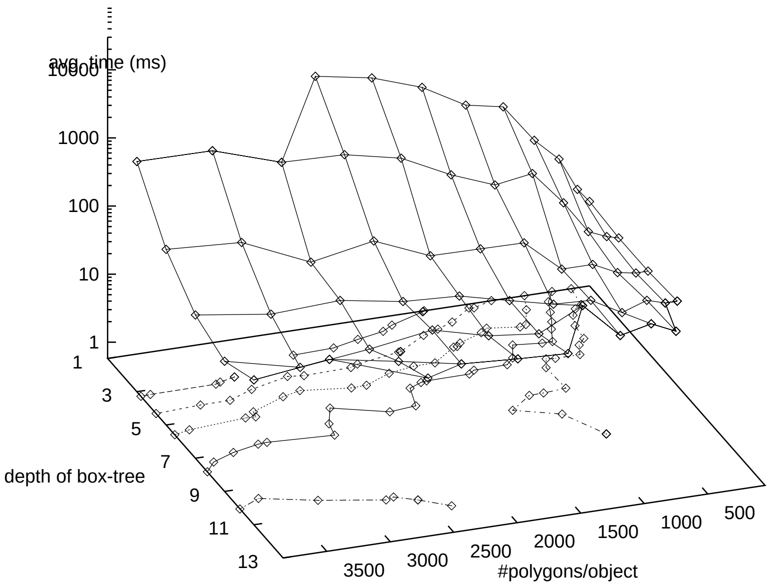


Figure 3.33. Same as Figure 3.32, but with two tori.

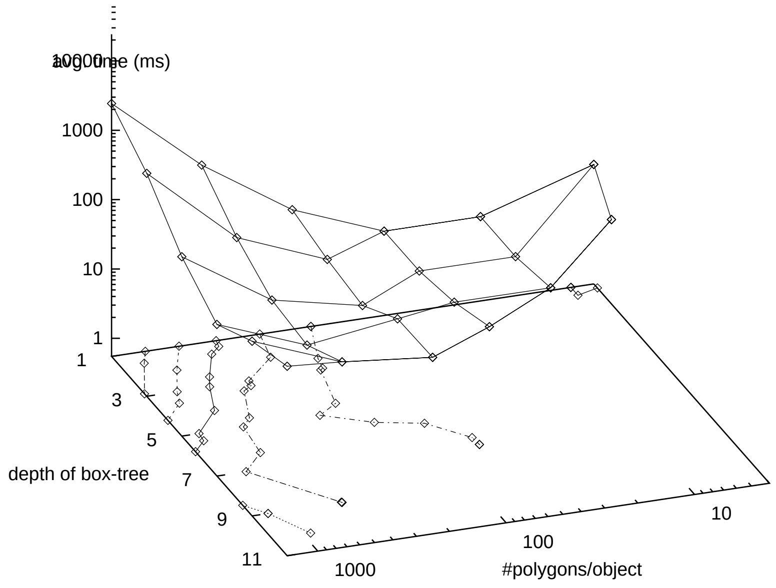


Figure 3.34. Same as Figure 3.32, but with two tetra-flakes.

3.6.6 Other Intra-Object Hierarchies

Box-trees are one way of grouping polygons hierarchically according to closeness. There are other possibilities; in fact, any bounding volume could be used instead of boxes (see Section 5.2). One author tried sphere-trees [Hubbard93]. I think they are not as well suited as boxes, since spheres usually have to overlap quite a bit if they are to cover an object (see Figure 3.38). The consequence is that many polygons have to be stored many times with many spheres — I suspect many more times than in box-trees. In contrast, a box-tree is a partitioning of the bounding box; thus polygons are inserted in several boxes only if they are cut. Furthermore, the algorithm which builds box-trees tries to minimize this number of cut polygons; which is not possible the way [Hubbard93] builds the sphere trees.

Also, constructing a sphere-tree doesn't seem to be as simple as constructing a box-tree: [Hubbard93] first builds an octree of the object, then encloses voxels by spheres, which will become the leaves of the sphere-tree. [OB79] constructs sphere coverings for objects which are given by their vertices.

As with space partitioning (see Section 5.3), one can also use a regular grid to partition an object's bounding box [GASF94]. But usually, hierarchical schemes outperform their regular flat counterpart, if they don't have to be re-built dynamically.

3.7 Other Approaches

This section describes some algorithms which have not been implemented, yet. Also, they don't seem to be as promising as the previous algorithms did. Still, I include them in this section, because they show other interesting ways how to look at the

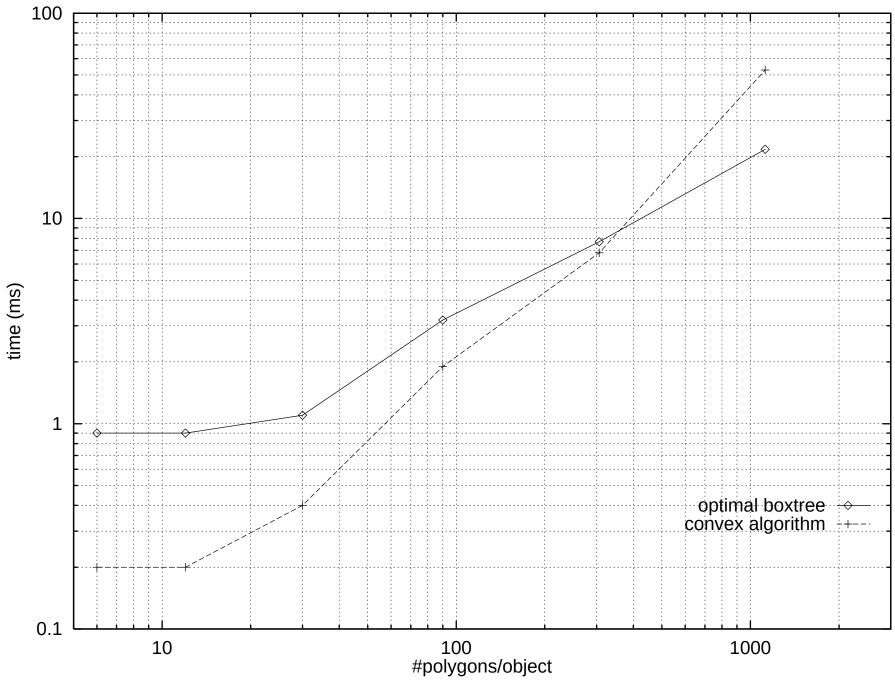


Figure 3.35. Comparison of the box-tree algorithm with the best conventional algorithm: Two spheres

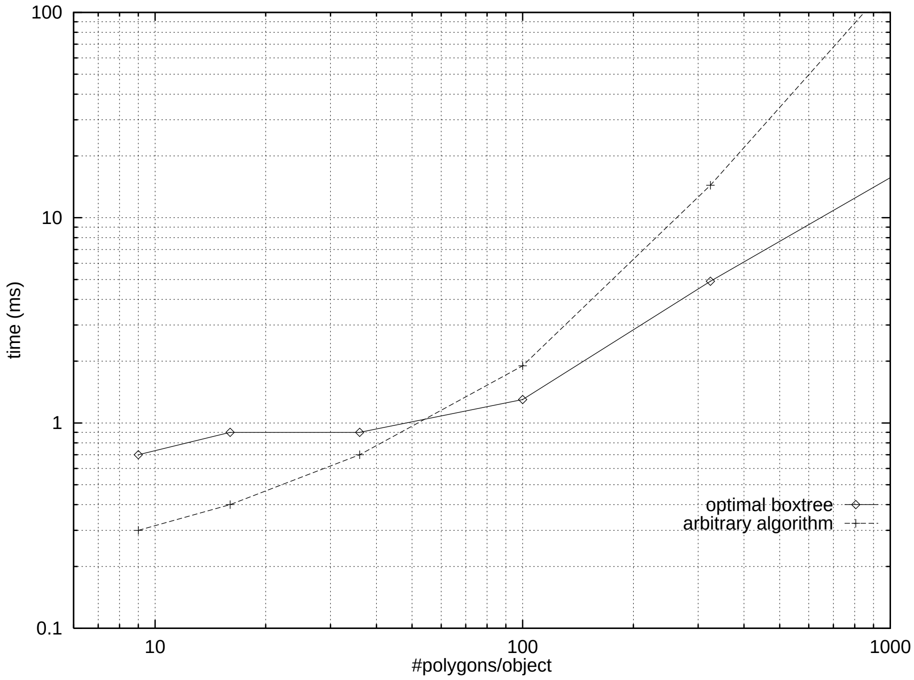


Figure 3.36. Comparison of the box-tree algorithm with the best conventional algorithm: two tori

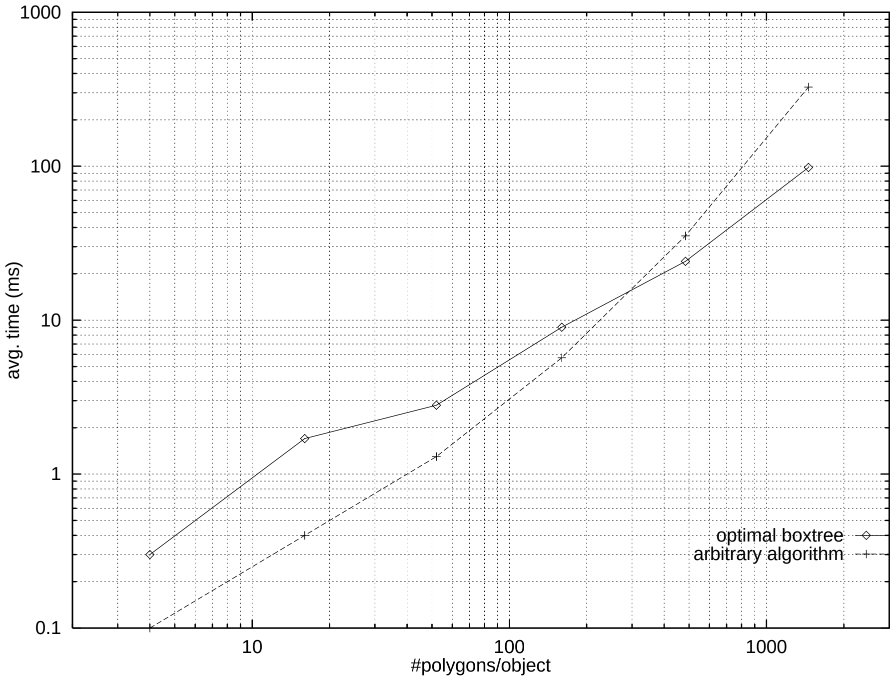


Figure 3.37. Comparison of the box-tree algorithm with the best conventional algorithm: two tetra-flakes

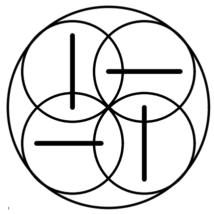


Figure 3.38. A sphere tree contains many spheres which overlap each other.

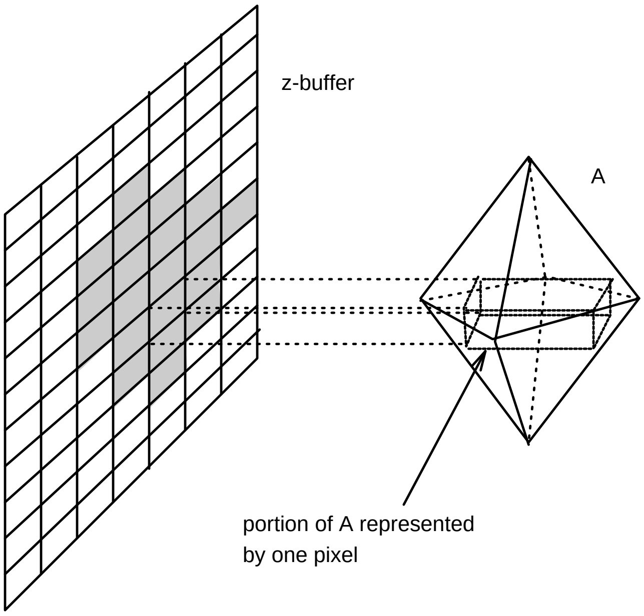


Figure 3.39. Discretization of a polyhedron by drawing it in the z-buffer.

problem from a different side, and thus help understanding the problem better. Unless these algorithms have not been implemented, one cannot be really sure, of course, that they don't offer advantages or speed-up.

3.7.1 Using the Z-Buffer

Using graphical hardware is always worth a try, since graphical hardware systems have gained very good performance. So, if we can use graphical hardware (namely the z-buffer) for collision detection, we might outperform any pure software solution.

Let's assume for a moment, that each pixel in the z-buffer can hold two depth values, z_{min} and z_{max}. Let's further assume that polyhedra are convex.

Given two convex polyhedra A and B, we can discretize A (conceptually) by projection onto the $z = 0$ plane. Every pixel will be "hit" exactly twice (A being convex); we will store the two depth values in the *minimum* and *maximum* depth value for every pixel. Thus, the image of A in the two-valued z-buffer represents a discretization (see Figure 3.39) by a grid with cells of size $[x, x + \delta_{\mathrm{pxl}}] \times [y, y + \delta_{\mathrm{pxl}}] \times [z, z + \delta_z]$. Here, δ_{pxl} is the size of a pixel, and $+\delta_z$ represents the resolution of the z-buffer along the z-axis. Depending on the application, we might want to choose an orthogonal or a perspective projection; in the latter case, the grid will not be regular, since the z-resolution changes with depth.

In order to check for a collision of A and B, we project all polygons of B onto the z-buffer; however, instead of drawing B, we just check for each pixel "hit" by B whether the color is that of A, and , if so, whether $z_{\mathrm{min}} \leq z_B \leq z_{\mathrm{max}}$. If we find such a pixel, then A and B do collide, otherwise they don't.

We can also find some measure of the depth of intersection. We just have to project all polygons of B and look for the pixel realizing

$$\max \{ \max\{ |z_{\mathrm{min}} - z_B| , |z_{\mathrm{max}} - z_B| \} \ : \ \textit{pixel hit by } B \}$$

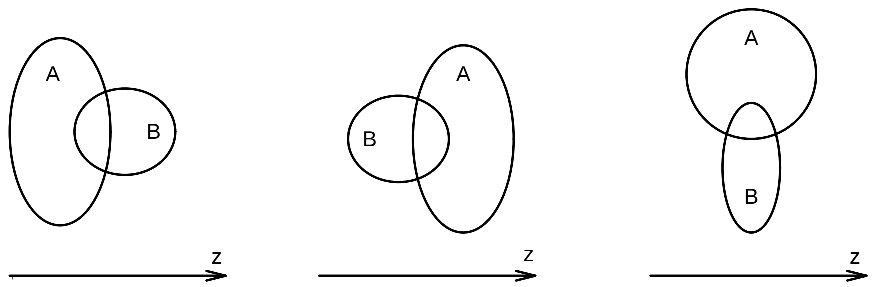


Figure 3.40. All possible arrangements of two convex objects.

Conventional z-buffers do not provide two depth values per pixel. The algorithm sketched above can still be implemented with a one-valued z-buffer, if it provides the following primitive drawing operations per pixel:

1. conventional (draw if depth of new pixel is less than the depth already stored),
2. draw only if depth of new pixel is less *and* color of new pixel is different from color already stored,
3. *erase* if color of new pixel is the same as the one already stored.

With these operations, we can simulate the algorithm above by 2 passes over all polygons of objects A and B.

1. Clear the z-buffer.
2. Draw all back-facing polygons of A conventionally (mode (1)).
3. Draw back-faces of B conventionally.
4. Draw front-faces of B using mode (2).
5. Draw front-faces of A using mode (3).

This will handle all three cases of possible arrangements of A and B as depicted in Figure 3.40.

Finally, the color buffer has to be scanned for any remaining color "B"; if there are pixels with that color, then a collision of (the discretized) objects A and B has occurred.

Of course, this scan should be eliminated if the hardware solution was to be really fast. This could be achieved very easily, since the only "feedback" from the hardware would be a single bit. Even with many fill processors this wouldn't be a problem; they just had to synchronize a write into this single bit, so as to prevent a concurrent write.

The algorithm could be augmented to be able to handle the larger class of closed polyhedra (not necessarily convex), if pixels could hold an arbitrary number of depth values.

The algorithm would work as follows: Project all polygons of A and store all z-values with pixels. After that, sort the z-values for each pixel (this could be done "on-the-fly" by insertion sort). Now, each pair of consecutive depth values (one odd-numbered, the other even-numbered) stored at the same pixel represent a portion of object A.

Then, project all polygons of B. If any depth value of B is between two consecutive depth values of A, then there is a collision.

3.7.2 Using a Convex Algorithm for Non-Convex Objects

We could use one of the algorithms above for convex polyhedra to detect collisions between non-convex polyhedra, too. To this end, we would have to partition each non-convex polyhedron into c convex parts [CP90a, RS92a, BD92b, SN86a], then we would check all c^2 possible pairs of parts (without loss of generality, we assume that both non-convex polyhedra consist of the same number c of convex parts).

The number c of convex parts can be quite high, at least theoretically. In practical cases though, I suppose $c \in O(N)$. Let N be the number of *reflex edges* (simply put, these introduce non-convexity; sometimes, they are also called *notches*). Then there are objects which cannot be decomposed into less than $O(N^2)$ convex pieces [Chazelle84a]. Actually, the problem of partitioning a polyhedron into a minimum number of convex pieces is NP-hard.

First, let's consider the average case. Let $t_c(n)$ denote the time used by an algorithm for collision detection between two convex polyhedra, each with n edges. If we assume that the non-convex object P can be partitioned into N convex pieces[7], and if we further assume that each convex piece consists of about $\frac{|E_P|}{N}$ polygons, then the collision detection between two such non-convex objects will take $O(N^2 t_c(\frac{|E_P|}{N}))$ time. Let's assume that $t_c(n) \in O(n)$. Then the overall time for collision detection of two non-convex polyhedra, will take $O(N|E_P|)$ time.

This means that this approach will be faster than using a non-convex algorithm, if the time of the convex algorithm $t_c(n) < \frac{1}{N} t_a(n)$, where n is the complexity of the polyhedra (in all practical cases $n \in O(|E|) = O(|F|) = O(|V|)$), and $t_a(n)$ is the time of the non-convex algorithm.

Drawbacks. This kind of approach has its drawbacks: Polyhedra need to be *closed* so they can be partitioned, whereas many models output from CAD systems do not meet this requirement, especially data coming from architecture systems, because these systems group polygons according to quite different criteria.

Also, partitioning polyhedra into convex pieces introduces an *alternative representation*, at least at run-time, maybe even in external files. Since this partitioning is a pre-process, any application which needs to *modify the geometry* while still doing collision detection among these objects cannot use this approach.

Algorithms for convex partitioning are non-trivial (to implement) [Chazelle84b, BD92a, SN86b]. Furthermore, the problem of finding a minimum number of convex pieces is NP-hard [RS92b]; although, there is an asymptotically optimal algorithm for polyhedra with zero genus [CP90b].

[7] The arm of a chair is a non-convex object, for which this assumption holds.

Chapter 4

B-Rep Data Structures

In most cases, the very simple b-rep data structure, namely vertex and polygon lists, will suffice for simple algorithms. But as soon as we want to improve algorithms, classify topology, exploit vicinity, or modify the b-rep, we are very likely to wish for a richer data structure. This happened to be the case with certain optimizations (see Section 3.2, page 31, for example) and for the first "real" application, the Y-Potter (see Section 8).

As usual in computer science, data structures and algorithms depend on each other (sometimes they are, in fact, just different ways to look at a problem). Richer data structures provide greater opportunities for more efficient algorithms.

4.1 Data Structures for B-Reps

The first additional list usually wanted is an edge list. Many algorithms presented in this thesis (on the pairwise level) use edges and polygons as basic features of a polyhedron.

Adding an edge list provides a big opportunity to build a data structure which contains information about adjacency and incidence. There are many ways to store adjacency and incidence [Woo85], [Weiler85]; a few of them will be reviewed briefly.

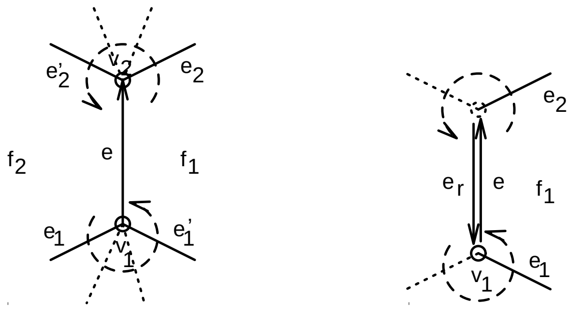
Following [Weiler85], we will refer to adjacency information as the *topology* of the polyhedron, whereas actual point locations, normals, etc. will be called the *geometry*.

All of the following data structures assume the polyhedron to be manifold and closed (see Section 4.2.1). This is because an edge always stores exactly two vertices and two polygons. If we would allow an edge to store k vertices and k polygons, we could probably represent k-manifold polyhedra[1].

The data structures to be presented below will be illustrated by Figures 4.1 and 4.2. The convention used there is that solid features (vertex, edge, polygon) are actually stored, whereas dotted features are just drawn to remind the reader that there are more of them in the polyhedron. The dashed arrows indicate circulation direction (conventionally counter-clockwise).

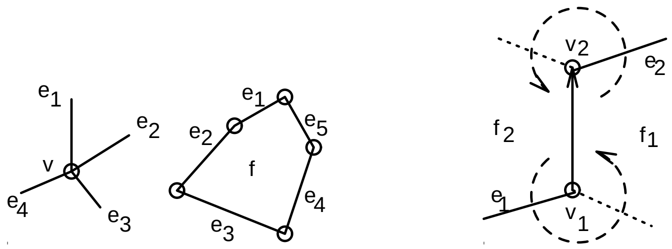
Winged-edge. This structure has been developed by Baumgart about 20 years ago. An edge stores its two vertices and the two incident polygons of which it is on the boundary. Additionally, each edge stores references to four adjacent edges: when

[1] How would adjacency information have to be stored? would we need to store neighbor edges in a certain order? if so, in which?



(a) winged-edge (b) quad-edge

Figure 4.1. B-rep data structures. Solid items are stored with the edge e, dashed/dotted ones are not.



(a) complete vertex/edge lists (b) doubly-connected edge list

Figure 4.2. B-rep data structures.

looking from "outside", they are the next edges in counter-clockwise and clockwise order around the vertices (see Figure 4.1(a)). Each vertex and each polygon stores an arbitrary pointer to one of its incident edges.

The winged-edge data structure needs $8 \cdot |E| + |V| + |F| \approx 10 \cdot |E|$ memory.

Quad-edge. This data structure allows accessing the same information with the same time complexity as the winged-edge structure. Instead of storing both "sides" of an edge, edges are stored twice in both direction, each directions stores half of the information [MS85b] (see Figure 4.1(b)).

It is not as memory efficient as the winged edge structure: $2 \cdot 5 \cdot |E| + |V| + |F| \approx 12|E|$. [MS85b] point out that this data structure allows for clockwise *and* counter-clockwise loops around vertices or polygons (in contrast to the DCEL, below) — however, this is also true for the winged-edge structure.

Complete edge lists. Each edge stores just its two vertices and its two incident polygons. Each vertex and each polygon has a list of all edges which are incident to it (see Figure 4.2(a)).

The disadvantage is that these individual edge lists can be of very different length, which causes potentially less simple code.

Like the DCEL (see below), it uses least memory: $2|E| + 4|E| + 2|E| = 8|E|$ [2].

In a real implementation, however, we would also have to store the number of elements per edge-array with every vertex/face.

[2] each edge stores 4 pointers $\rightarrow 4|E|$; each vertex stores 1 pointer per edge, each edge-pointer occurs exactly twice in the vertex list $\rightarrow 2|E|$; analogously for faces

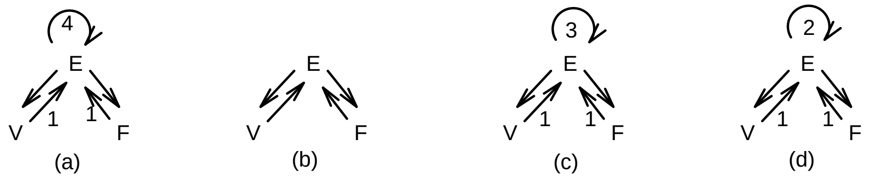


Figure 4.3. The winged-edge (a), complete vertex and edge list (b), quad-edge (c), and doubly-connected edge list (d), depicted by the relations they store.

Doubly-connected edge list. This is the "poor man's" version of the winged-edge data structure: edges contain two edge pointers less (see Figure 4.2(b)), which makes this data structure as memory efficient ($8|E|$) as the complete-edge-lists. Plus, it provides much more uniform access to polyhedral features.

4.1.1 Classification

The data structures given above can be classified according a scheme of [Woo85].

Let $P = (V, E, F)$ be a polyhedron (graph), and $\mathcal{V} = \mathcal{P}(V)$, $\mathcal{E} = \mathcal{P}(E)$, $\mathcal{F} = \mathcal{P}(F)$. Then P induces relations out of $V \times \mathcal{V}$, $E \times \mathcal{E}$, $F \times \mathcal{F}$.

We write $X \rightarrow Y$ if the relation induced by P is completely stored in a data structure. If the structure contains only n $y \in Y$ related to every $x \in X$, then we will write $X \xrightarrow{n} Y$.

With this notation we can depict the data structures above as shown in Figure 4.3.

4.1.2 The DCEL

In the current implementation, the doubly-connected edge list (DCEL) was chosen to store topology. Actually, it does contain a little bit more than the pure doubly-connected edge list: it contains also the full relation $F \rightarrow V$, i.e., each polygon has got an array of all its incident vertices. This is needed by the renderer.

It is not quite as fast as the winged edge if there are many applications which need to go from one edge to the next in clockwise *and* in counter-clockwise fashion. Yet, this seems to be rarely needed (it was not needed up to now) — and the doubly-connected edge list is more memory efficient.

This data structure allows (as do the others above) to enumerate all incident edges of a vertex in linear time (which is optimal):

```
Loop around vertex

input:  v
e0 := arbitrary incident edge incident to v          {stored with v }
output e0
e0.v1 == v   ⟶    e := e0.e1
 e0.v2 == v   ⟶    e := e0.e2
 while e ≠ e0
    output e
    e.v1 == v   ⟶    e := e.e1
     e.v2 == v   ⟶    e := e.e2
```

Quite similar, we can loop over all edges on the border of a certain polygon (note that a loop around a vertex is in counter-clockwise order, whereas a loop around a polygon is in clockwise order). Furthermore, we can loop over all neighbor vertices of a certain vertex or over all neighbor polygons of a certain polygon — simply by adding one line to the above function.

4.1.3 Building the DCEL from the Input File

At start-up time, we are given only the relation $F \rightarrow V$, i.e., a bunch of polygons, each consisting of a list of vertex numbers. We want to construct from this rather meagre data structure the richer doubly-connected edge list.

There are a few things which complicate this task, but which we have to deal with: the polyhedron might not be 2-manifold, it might contain polygons which are oriented wrong (inside out), it might not be closed, or it might contain degenerate polygons.

The very basic idea for the construction of the DCEL is simply to loop over all polygons and over all their vertices, and check if the current edge is already in the edge-list. If it is, we add a few pointers to this edge entry; if it is not, we create the edge entry and stick in half of the pointers. The code itself is a little bit more complicated because we have to be able to detect all of the above mentioned "ugly" possible inputs — and then deal reasonably with them.

One step of the algorithm is the check whether an edge has already been created in the edge list. The naive approach is to search the edge list built so far. This introduces, of course, quadratic complexity.

A better solution is to keep temporary lists of neighbor vertices with each vertex. So, when we want to check if an edge (v_1, v_2) has already been created, we just have to scan the edge lists of the two vertices v_1, v_2. Whenever a new edge is added to the edge list, we add each of the two vertices to the neighbor list of the other vertex.

This decreases run-time tremendously; at least for polyhedra with bounded vertex-degree it is now linear in $|E|$. Timing of the building of the DCEL for a polyhedron with 1220 edges showed the following result: the naive algorithm took 2.8 sec, the linear algorithm took only 40 msec (on an R4400, 50 MHz).

Additional memory requirements are only temporary, and they are as low as $2|E|$ (for the vertex neighbor lists).

4.2 Topological/Geometrical Properties

If we want to exploit topological and geometrical properties of polyhedra for collision detection, we first have to determine them. Topological properties we will consider are

- closedness (we can define "interior" and "exterior"),
- manifoldness (each edge is on the boundary of exactly two polygons),
- orientedness (all polygons "face" in the same direction),

whereas geometrical properties are

- all polygons are planar,
- all polygons are convex,
- the polyhedron is convex.

Some of them were mentioned in Section 1.2.2, page 9. All of them can be determined by functions within the Y system.

Some of the algorithms of Section 3 require certain properties; for example, closedness is a prerequisite of the algorithm of section 3.5, convexity is a prerequisite of many algorithms, like the one in Section 3.4, manifoldness is needed by the relaxed polygon collecting phase (see Section 3.2, page 31).

4.2.1 Topological Properties

Manifoldness. The *manifoldness* (strictly speaking: 2-manifoldness) of a polyhedron P is defined as follows:

$$\begin{aligned}
&P \text{ manifold } :\Leftrightarrow \\
&\forall p \in \partial P\ \exists \varepsilon : U_\varepsilon(p) \cap P \text{ topologically equivalent to a disc} \quad \Leftrightarrow \\
&\exists \text{ bijection between } U_\varepsilon(p) \cap P \text{ and } \{x \in \mathbb{R}^2 \mid \|x\| \leq 1\}
\end{aligned}$$

Fortunately, for polyhedra given as B-reps, the condition for being manifold is much simpler: the only places which could introduce non-manifoldness are edges. So, we "just" have to scan all edges and see if one of them exists more than two times in the vertex lists of all polygons.

This, of course, is way too expensive — the complexity is at least quadratic. The doubly-connected edge list helps here: while building it, we get the property "manifold" as a by-product! Each time we process an edge of a polygon (which was derived from the vertex list of the polygon), we either add it to the edge list, or we add some pointers to the already existing entry in the edge list. It is during this pointer-adding step, where we can detect non-manifoldness very easily: if all the pointers in an edge record have been filled in already, then that polyhedron can't be 2-manifold (because we are processing the $\geq$3rd occurrence of that edge right now).

Closedness. The test for closedness is as easy as the one for manifoldness. The only difference is that we have to build the DCEL first, before we can do the check. A polyhedron is not closed if it contains an edge which is on only one polygon's border. We can test for that by simply scanning all edges in the DCEL; an edge with less than two incident polygons can be recognized by an incomplete pointer record.

Non-closedness and non-manifoldness are not mutually exclusive, of course.

Orientedness. We can detect whether there is a polygon which has the wrong orientation while building the DCEL; we cannot exactly determine which one, but we can determine *two* polygons, *one* of which is wrong.

If there is a polygon within the polyhedron oriented the wrong way, then we will detect this because of certain pointers having wrong values assigned to them. Figure 4.4 depicts the two possible wrong pointer assignments. Case (a) is produced when the edge is created while processing the wrong oriented polygon, case (b) is produced if the wrong oriented polygon is processed second. Both of them can be detected if we consider the face- and the vertex-pointers of the edge.

The advantage of the topological approach is that it works just as well for non-closed polyhedra, whereas a geometrical approach would depend on the notion of some "interior" (probably determined by the barycenter). So, the topological approach can be applied to a much larger class of polyhedra. Of course, when *all*

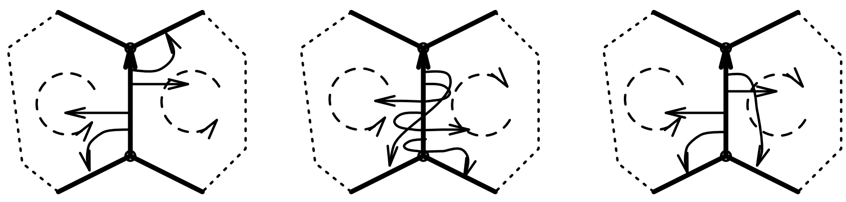


Figure 4.4. The function building the DCEL will produce wrong pointering if a polygon of the input polyhedron is oriented the wrong way.

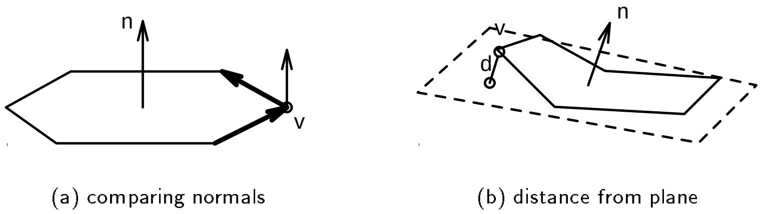


Figure 4.5. Two methods to test flatness of a polygon.

polygons are oriented wrong, it doesn't find *anything*; also, it cannot decide which polygon is wrong — that must be based on some "majority" decision.

4.2.2 Geometrical Properties

These properties can be determined by just considering the location of vertices in space. Convexity, however, is much easier to determine, if we are given a complete DCEL (see below).

Flatness. All vertices of a polygon have to be in the same plane (coplanar). There are a few, rather similar ways, how to determine this, two of them are sketched below.

The first method (see Figure 4.5(a)) calculates at each vertex a local normal (based on the vertex itself and a "few" others). Then it compares this normal to a reference normal, either the pre-computed face normal or the first vertex normal. The comparison can be a cross product, or a "distance" between the two normals.

The second one (see Figure 4.5(b)) computes the distance of each vertex from the plane which is given by the face normal and a point on the polygon (e.g., one of its vertices).

The only difference seems to be a slightly better numerical condition with method (b) of Figure 4.5, because it doesn't need to compute a cross product at each vertex. A dot product (used for the distance computation) is usually faster, too. Therefore, I implemented the second algorithm in the Y system.

Convex polygons. We could test for this property by considering vertex normals (computed "locally"), too. However, this approach has some disadvantages: computation of those local normals is prone to numerical errors, even more so, because there might be polygons with very small angles or angles close to 180°. Furthermore, we have to know the proper orientation so we can compare these vertex normals to some reference normal (see Figure 4.6).

Altogether, this doesn't seem to be too appealing, because convexity doesn't have anything to do with orientation of polygons. (Besides, the cross product is slow.)

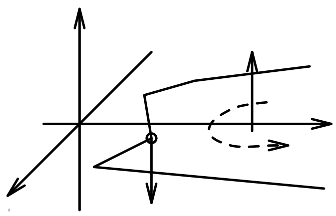


Figure 4.6. An expensive, less robust method to test polygons for convexity.

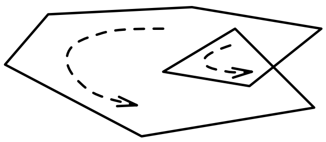


Figure 4.7. The robust method can also discover non-convexity of self-overlapping polygons.

The approach taken here is solely based on the fact that the sequence of y-coordinates and the sequence of x-coordinates of a convex polygon have got exactly two extrema. This holds for 2D polygons as well as for 3D polygons.

This approach has two advantages:

- it uses only simple floating point comparisons, and doesn't depend on the ordering of vertices ("clockwise or counter-clockwise?").
- it is a global test, i.e., it can detect non-convexity of polygons with self-overlaps (see Figure 4.7).

However, both methods cannot deal with degenerate polygons, i.e., polygons which do not have any interior (they should be classified non-convex).

Convexity. Convexity of polyhedra is also a property which does not bear any connection with ordering of vertices. We could determine convexity by constructing the convex hull of the polyhedron and comparing that with the original one. However, this approach seems to be too involved in order to be implemented in a day.

Another approach is to use the information obtained so far, and then check that every dihedral angle is smaller than 180°, i.e., that the edge is not a reflex edge. Angles are measured along the inside of the polyhedron, so we need to know all face normals, which should have the same orientation (we assume here that they point outward).

The dihedral angle at an edge $e = (v_1, v_2)$ between two polygons f_1, f_2 can be computed by (see Figure 4.8)

$$\begin{aligned} &\frac{1}{2}\cos(\measuredangle(f_1, f_2)) = n \cdot n_1, \qquad \text{, where} \\ &n = (v_2 - v_1) \times (n_1 + n_2) \end{aligned}$$

Checking only dihedral angles is sufficient if the following prerequisites hold:

- the polyhedron is manifold and closed (it doesn't make any sense to define convexity for non-manifold, or non-closed polyhedra),
- all polygons are planar; checking only dihedral angles would classify "Schönhardt's prism" [Schönhardt28], shown in Figure 4.9, as convex,
- all polygons are convex; this is not a necessary prerequisite, because the above algorithm would find out anyway, but it's a trivial pre-check (the current implementation does a full classification, anyway).

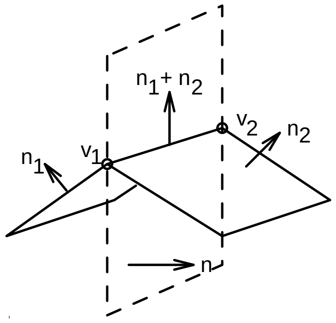


Figure 4.8. The dihedral angle can be computed by cutting it in half.

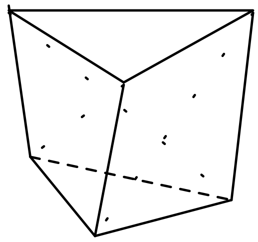


Figure 4.9. All dihedral angles are convex, no matter which definition of face normal we use. (Triangulating the quadrilaterals along the dotted lines would introduce notches.)

Chapter 5

Space Partitioning Methods

5.1 Need for Space Partitioning

Highly dynamic environments (e.g., virtual environments), where many objects should be checked against each other for collision, pose the *all-pairs problem* (see Section 1.2.2) on the global level — just like the all-pairs problem on the edge-face level. Even if only a few objects are to be checked against many other static ones, we don't want to check all possible pairs. Of course, before doing any collision check between two objects, we first check their bounding volumes for intersection. Still, the complexity is quadratic, i.e., with n moving objects we have to do $\sim \frac{n^2}{2}$ bounding volume checks.

The basic idea with all space partitioning methods is to exploit *space coherency*: most regions of the "universe" are occupied by only one object or empty. Consequently, each object has a very small number of "neighbors"; only these neighbors have to be tested for collision with the object itself. The difference among approaches is the data structure, which is maintained during run-time.

The serious constraint is the dynamic quality of the environment. While the only criterion with static environments is *fast retrievability* or *fast neighbor-finding*, the criterion with dynamic environments is, in addition, *fast updating* for moved objects.

For reasons made clear in the next section, we will deal only with bounding volumes throughout this whole chapter. So, whenever we use the term "object" in this chapter, we mean its bounding volume.

When is space partitioning worth the effort? One might ask, what conditions must be met so that any kind of method which tries to avoid testing all n^2 object pairs is worth the overhead.

Let $T_s(n)$ be the time spent to update n objects in the global data structure plus the time to find all object pairs which have to be tested further for exact collision. Let $T_p(n)$ be the time spent for one exact collision test of one pair (we assume all times to be averages). Then a global phase will be efficient if the following relation holds for the average number $p(n)$ of object pairs tested further for exact collision:

$$p(n)T_p(n) + T_s(n) < \frac{n^2}{2}T_p(n)$$

So the average number of exact collision tests per frame should be

$$p(n) < \frac{n^2}{2} - \frac{T_s(n)}{T_p(n)}$$

5.2 Bounding Volumes

Bounding volumes are used in almost every area of computer graphics to speed up computations by not doing them at all if the result can be obtained easily by a pre-check.

So, even without any space partitioning method, one does want to do a bounding volume pre-check before doing any further collision detection. In the context of space partitioning, we want to deal only with bounding volumes, too, because dealing with the objects themselves is usually way too expensive.

In order to gain any speed, the bounding volume must be much simpler an object than the object it bounds. At the same time, this is the general disadvantage of bounding volumes: depending on the geometry of the object "inside", they could contain very much "empty" space.

There are a few very simple, commonly used bounding volumes:

- Simplest of all is the *axis-aligned bounding box.* Its faces are always parallel/perpendicular to the world coordinate system. Many pre-checks using axis-aligned bounding boxes are just comparisons of coordinates, which make them very fast.

  They have some disadvantages, however. The first one depends on the method how they are computed: either they are computed from scratch every time the object has rotated (i.e., do a whole pass over all vertices in world coordinates), or we transform (conceptually) all eight vertices of the box, and then put an axis-aligned box around those. The problem with the first approach is that it is very expensive; the second one is fast, but unfortunately, the boxes generated by it are up to 2 times as large in volume as the original bounding boxes (see Figure 5.1(a)).

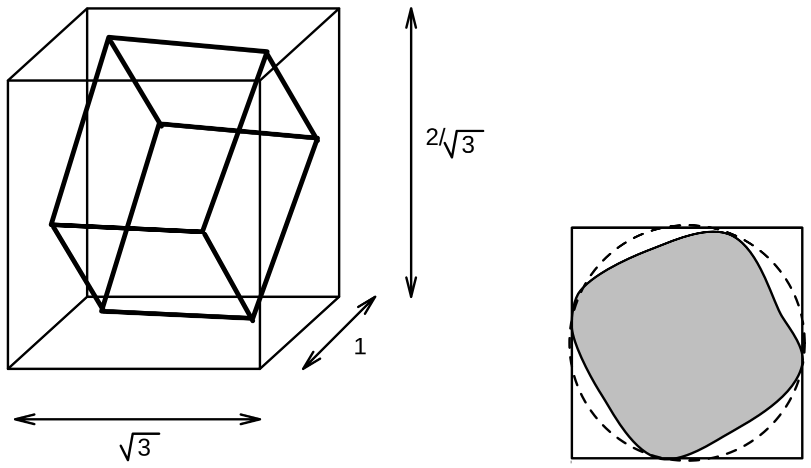
  The second disadvantage depends on the geometry of the object "inside": the more "spherical" the object is the less tight the bounding box can be (see Figure 5.1(b)). A box around a sphere is $6/\pi \approx 2$ times larger in volume than the sphere.

- *Spheres* are second in popularity. The geometry of spheres is the simplest one, which makes them attractive. The transformation of spheres is very simple, too, since we only have to transform one point; there is no problem with axis-alignment.

  An overlap test between two spheres is not quite as inexpensive as an overlap test between two axis-aligned boxes[1], but it is cheaper than a test between two arbitrarily oriented boxes. Overlap tests between spheres and axis-aligned boxes are fairly cheap, too [Kirk92a].

  Computing the optimal bounding sphere is not nearly as easy as computing a bounding box — the brute force method takes way too long for more than 20 points! The algorithm usually chosen is *linear* or *quadratic programming* [Megiddo82, EH72]. Computing an almost-optimal solution seems to be more feasible [Welzl91, Glassner90b, Kirk92b].

[1] especially if we bear in mind that most box-overlap tests are finished after one or two float comparisons



(a) aligned box around bounding box can be 2 times larger

(b) lots of empty space inside box around spherical objects

Figure 5.1. Disadvantages of boxes as bounding volumes.

- *Slabs* have gained quite some popularity with ray tracers [KK86]; in fact, they are a generalization of bounding boxes. A slab is defined as the (infinite) space between two planes; the conventional bounding box is then just the intersection of three slabs. Consequently, the idea is to enclose an object not only by three slabs, but by (potentially) arbitrarily many (see Figure 5.2).

  If we use the same set of slabs for every bounding volume, then a lot of computational work can be saved, because with many tests (like ray-bounding-volume intersection) we can pre-compute and store in tables many intermediate results.

  Like with bounding boxes, two floats are sufficient to store one slab.

  However, it is not clear (to me) how an efficient transformation of slab volumes would work, since the vertices of the volume are not stored; calculating them for each transformation would be way too expensive, too. This explains probably, why they have not been used (yet) in dynamic environments, but only for ray tracing.

- *Cylinders* seem to be simple, too; however, they don't seem to be too useful, even with static scenes, probably because their geometry is already too complicated [WHG84].

Summarizing, some of the desirable characteristics of bounding volumes are:

- easy to compute,
- little memory requirements,
- fast transformable,
- simple overlap check,
- tight fitting.

Bearing in mind the above discussion of bounding volumes, I decided to use axis-aligned bounding boxes first (although, an experiment with non-aligned boxes was realized, too). Since testing for an overlap between an axis-aligned box and a sphere is fairly inexpensive, too, it should be easy to carry over all algorithms developed for axis-aligned boxes in the current chapter.

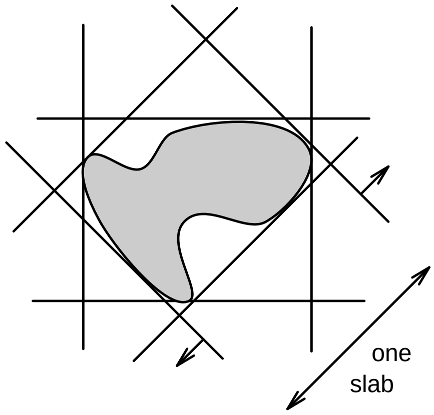


Figure 5.2. Bounding volume consisting of slabs (pairs of parallel planes).

5.2.1 Efficient Transformation of Boxes

There are essentially two ways to represent an axis-aligned box:

- by specifying two diagonally opposite corners, which is commonly stored as $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}]$.
- by specifying a "low" corner and the vector to the diagonally opposite corner.

Of course, both forms are trivially transformable into each other. In different situations they provide different possibilities for an efficient implementation.

The very naive transformation of axis-aligned bounding boxes would transform all 8 vertices, and then determine for every axis which of them has got the maximum/minimum coordinate.

The more efficient approach takes advantage of the special situation [Glassner90a]. We will consider the calculation of $x'_{\max}$, the maximum x-coordinate of the transformed bounding box, given by the first representation (two corners). Let's call the transformation matrix M; let $|(\cdot)|_x$ be the x-component of the vector $(\cdot)$. We will give the derivation here only for two-dimensional boxes — an extension to higher-dimensional ones is straight-forward. $x'_{\max}$ can be computed by

$$
\begin{aligned}
x'_{\max} &= \max\{\, |(x_{\max}, y_{\max})M|_x,\ |(x_{\max}, y_{\min})M|_x, \\
&\qquad\quad |(x_{\min}, y_{\max})M|_x,\ |(x_{\min}, y_{\min})M|_x \,\} \\
&= \max\{\, x_{\max}M_{11} + y_{\max}M_{21},\ x_{\max}M_{11} + y_{\min}M_{21}, \\
&\qquad\quad x_{\min}M_{11} + y_{\max}M_{21},\ x_{\min}M_{11} + y_{\min}M_{21} \,\} \\
&= \max\{\, x_{\max}M_{11},\ x_{\min}M_{11}\} + \max\{y_{\max}M_{21},\ y_{\min}M_{21} \,\}
\end{aligned}
$$

by just applying the law $\max\{a+b, a+c\} = a + \max\{b, c\}$ twice. The other components of the transformed box can be computed quite similar. For the second representation of boxes (corner and diagonal), a similar calculation can be found.

The algorithm would be something like:

`Fast aligned bounding box transformation`

`input:` $a^{\min}, a^{\max}$ $\{$*the two corner points* $\}$
`output:` $b^{\min}, b^{\max}$

$$
\begin{array}{ll}
i = 1 \ldots 3: & \\
\quad b_i^{\min} = M_{4\cdot} & \{\textit{assumes } 4 \times 4 \textit{ matrix}\ \} \\
\quad b_i^{\max} = M_{4\cdot} & \{\textit{and “}v \times M\textit{”}\ \} \\
\quad j = 1 \ldots 3: & \\
\qquad x := a_j^{\min} M_{ji} & \\
\qquad y := a_j^{\max} M_{ji} & \\
\qquad b^{\min} := b^{\min} + \min(x, y) & \{\min \textit{ and } \max\ \ \} \\
\qquad b^{\max} := b^{\max} + \max(x, y) & \{\ \textit{cost only 1 comparison}\ \}
\end{array}
$$

This algorithm needs only 18 multiplications, 18 additions, and 9 comparisons, which is much more efficient in comparison to the 72 multiplications, 72 additions, and 48 comparisons of the naive algorithm.

5.3 Classification

Several data structures have been developed in the past 15 years (see [BF79] for an early paper); most of them were designed to represent single objects, but they can also be used for space partitioning.

BSP. Binary space partitioning was developed to partition a whole scene (a bunch of polygons), so as to solve the hidden surface problem [FKN80].

The idea is to cut space recursively into two halves by a plane; we start with the whole "universe", cut it in half, and continue with each of the halves (see Figure 5.3). For visibility pre-processing, each plane is chosen to be the supporting plane of a polygon of the scene. Then, for any given point p, we can traverse the tree in an in-order manner, thereby processing polygons back to front.

For visualization of dynamic scenes, the data structure has been augmented by so-called "auxiliary planes" [Torres90]. These try to divide space without cutting objects (see Figure 5.4).

Despite the simplicity of the basic idea, there is the annoying problem of cut objects and/or polygons. In general, this cannot be avoided, and the size of a BSP tree can get rather large — the lower bound is $\Omega(n^2)$ for n facets. Only recently an algorithm has been presented which can construct the optimal BSP (see [PY90]).

It seems to me that in the recent past, BSPs have been mainly used for solid modeling, where objects can be represented and operated on by using BSPs [TN87, NAT90].

Cell subdivisions

There are two basic classes of cell decompositions of space:

- uniform; associated algorithms are very simple and as such pretty fast. However, these data structures cannot adapt to large, entirely occupied or empty regions of space.
- hierarchical; advantages and disadvantages compared to the uniform structures are just swapped. In addition, they usually use less memory.

In contrast to BSPs, cell subdivisions are not "object oriented" but space oriented, i.e., the data structure itself does not depend on the arrangement of objects. (With BSPs, the choice of partitioning planes depends heavily on the current arrangement of objects.) Instead, this data structure is built once at start-up time; later on, cells are just "filled" with the objects they contain.

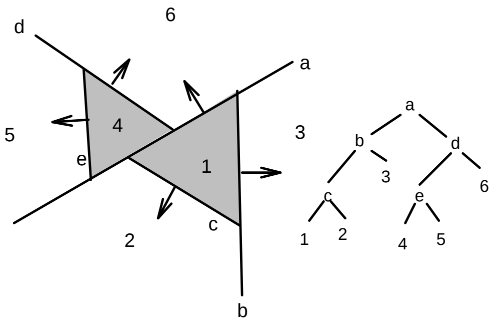


Figure 5.3. A BSP tree of set of polygons.

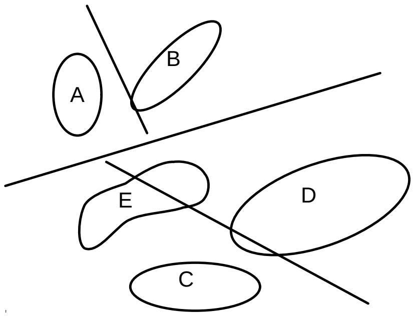


Figure 5.4. BSP trees with auxiliary planes for dynamic scenes.

Grids. Almost always regular grids are used, i.e., the cells are rectangular boxes (this is probably *the* simplest space partitioning structure). This makes them intriguing simple; hence, algorithms operating on regular grids are very simple, too.

For ray tracing, regular grids have proved to be the second best data structure [MSH+92].

Octrees. Octrees can be considered a multi-resolution grid; usually, only certain cells of each layer will be actually allocated memory for (see Figure 5.5). Some algorithms nevertheless refer to all cells on all layers implicitly by a certain labeling scheme [Sung91].

A thorough survey about octrees can be found in [Samet90b, Samet90a]. However, the description is restricted to static representations.

Octrees have been heavily used for ray-tracing to speed up the ray-object intersection test [GA93, MSH+92, Sung91]. Solid modeling is another area, where they are used for representation of objects [TKM84, FK85, NAB86]. Lately, the octree representation has been combined with the b-rep in order to combine the benefits of both: fast algorithms for boolean operations using the octree, and exact object representation (see also [CCV85]). Also, the two-dimensional brother (quadtrees) has been used for image compression [Samet90b, Samet90a].

In computational geometry, octrees and quadtrees have been investigated for solving the point-location and the range-query problem efficiently [PS85, BF79, YDEP89]

It has often been brought forward that octrees are very memory expensive. This might be true if several hundreds of thousands of polygons were to be stored in an octree; however, [Dyer82] shows that a quadtrees is a space efficient representation of boxes, precisely, that the average and worst case numbers of nodes in the quadtree are both in the order of the box' perimeter plus the logarithm of the box' diameter (see Figure 5.6).

Certain properties of objects can be computed quite easily by approximating them with octrees, e.g., the volume and the inertia tensor can be computed very simply with help of octrees. These properties are approximated the better the higher the resolution of the octree is; however, this is not necessarily true for all features of an object: surface normals and surface area stay the same no matter how finely resolved the octree is.

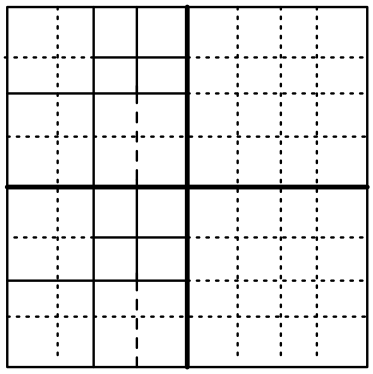


Figure 5.5. An octree can be regarded as a data structure for efficiently storing multi-resolution regular grids.

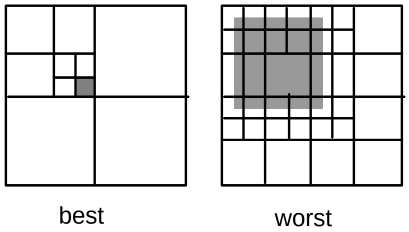


Figure 5.6. Best and worst case positions of a box with respect to memory usage.

Non-conventional space partitionings

Just to remind us that there are many other (maybe clever) algorithms, here are some less well-known.

Macro-regions. These are based on a (fixed) grid and, like octrees, the method tries to group large areas of contiguous empty voxels [Devillers88]. They are more flexible, though, because they do not superimpose other grid layers. Instead, these areas of empty voxels (called "macro-regions") are rectangular, and as large as possible. They may (and usually will) overlap (see Figure 5.7).

Sphere grids. Conventionally, grids are made of cubes. However, they could quite analogously be made out of any other bounding volume, too, for example, of spheres (see Figure 5.8). Inserting an object is exactly the same algorithm, except that the underlying overlap test of the bounding volume with a cell is different.

Exactly like an octree is built from a grid by superimposing coarser grids, an octree can be built with spheres (or other bounding volumes), too.

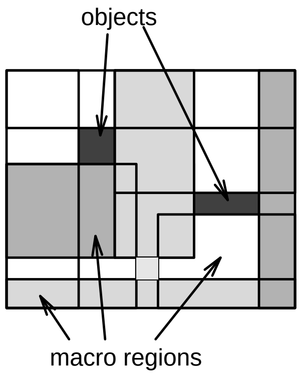


Figure 5.7. Macro-regions designate voxel rectangles which are empty.

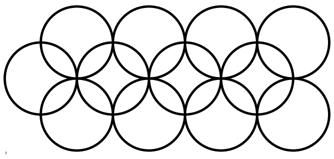


Figure 5.8. Instead of boxes, spheres could be used to build grids.

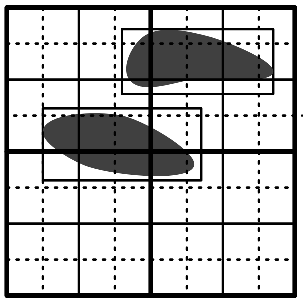


Figure 5.9. Objects might occupy several cells together, which could cause the same object pair to be generated more than once.

Octrees and grids have been implemented in the collision detection module; the module can be configured at run-time during initialization to use either of the two.

The bounding box hierarchy is (not yet) exploited. Two reasons led to this decision: the correlation between geometrical vicinity and vicinity within the hierarchy is probably very low in a highly dynamic environment (which is the interesting case for collision detection); in order to yield any significant improvements compared to the quadratic all-pairs case, the hierarchy must be well modeled and fairly deep — which does not seem to be the case with too many models (unless it can be done automatically).

Since ray tracing literature suggested octrees in favor over grids, I implemented them first, then grids for comparison.

5.4 Octrees

The basic idea with any cell decomposition in the context of collision detection is: whenever we want to know which other objects a given object could collide with, we don't have to consider object pairs which do not share a cell, i.e., there is no cell which does not overlap (partially) with the two objects' bounding box at the same time.

In order to find all object pairs which might intersect, we make a pass over all objects; for each object we look at all the cells it occupies. Then we have to do an exact collision detection only with those objects which are in one of these cells, too.

However, there is a little problem (which is similar to a problem with ray-tracing): two objects might occupy several cells at the same time (see Figure 5.9). With the naive approach, the same object pair would be handed to the exact collision detection several times in one frame. However, we can get over this problem very easily with the time-stamp technique (see Section 7.3.1).

Before we can look up object pairs in the space indexing data structure, we have to insert the objects. Then, a very simple enumeration of cells occupied by an object would be just to execute the insertion algorithm once more, but instead of inserting the object, we would just add all those objects which are already there to some list (conceptually).

When objects move in-between frames, the data structure has to be updated. This can be done again naively by just using the insertion algorithm once more: we make one pass with the old position to remove any reference to the object to be moved, then we make a second pass with the new position to insert it again.

5.4.1 Basic Insertion Algorithm

The basic algorithm for inserting an object (that is, in this chapter, a box) $\boldsymbol{b}$ is quite simple:

```
Octree insertion

input:  box b, octant o
o is leaf, i.e., level(o) = max  ⟶
        add b to o's list
      return
o completely inside b  ⟶
        add b to o's list
      return
i = 1...8:
b intersects with sub-octant o_i
  ⟶     insert b in o_i
```

See Section 7.3.4 for some remarks on an efficient implementation.

Since an octree is (conceptually) a multi-layer uniform grid, we can use integer coordinates for doing the box-octant comparisons. We convert world coordinates into integer coordinates, based on the finest grid, with the following formula (the octree has depth n):

$$\left\lfloor \frac{x - o^{\min}}{o^{\max} - o^{\min}} 2^n \right\rfloor$$

Note that this defines a voxel to be a "half-open" cube $[v_x, v_x + \delta_x] \times [v_y, v_y + \delta_y] \times [v_z, v_z + \delta_z]$. Of course, the factor $\frac{2^n}{o^{\max} - o^{\min}}$ should be pre-computed at initialization time.

(A little excursion: using integer coordinates gives rise to a simple access scheme when inserting points into the octree: if a point x has integer coordinates (x, y, z), then the bits x_i, y_i, z_i, read as a 3-bit number, are the number of the octant on level i which the point is in. In other words, the string $(x_n, y_n, z_n), \ldots, (x_0, y_0, z_0)$ is the path from the root to the leaf which contains the point [GA93].)

5.4.2 Finding "Nearby" Objects

After all objects have been inserted in the octree, the next step of the overall collision detection is to find all "nearby" objects to a given one. In this section (and Section 5.5) we will consider two objects *neighbors* if they *occupy some elementary cell (voxel) together.*

This can be done very simply by just using the insertion algorithm once more; except inserting the object, we just scan the list of "attached" objects. The list of "attached" objects is an *object array* at every octant node, which stores pointers to all objects which (partially) occupy the octant (see Figure 5.10). (See Section 7.3.4 for an efficient implementation of these arrays.)

The only difference to the insertion algorithm is the case when an octant is completely inside the box: with insertion, we're finished with recursion at this point; for finding neighbors, however, we have to descend further down the whole sub-tree rooted at this octant, since there might be objects which are attached only to some octant at a deeper level. Nevertheless, the two boxes are neighbors according to the definition.

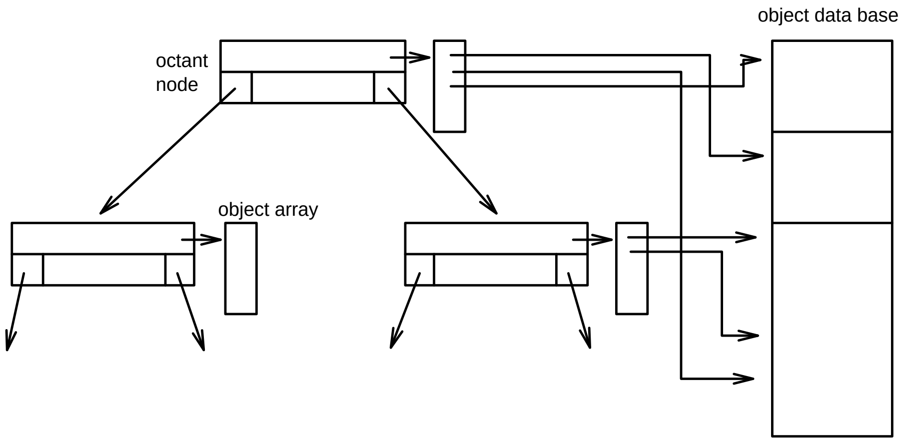


Figure 5.10. Octree data structure with object arrays at each node.

Octant arrays. We can do a little better by enhancing the data structure by "octant arrays"; every object will be given its own octant array. This array contains references to every octant whose object array has got a reference to the object in question (see Figure 5.11). When finding neighbors of an object, we don't have to traverse the octree from the top. Instead, we can start traversal at those nodes which are referenced by the object's octant array. (Most of them will be leaves; see Section 5.4.4.)

However, this data structure needs more memory — not only for the octant arrays, but also for 6 integers with each cell, which hold the cell's extent. That is about 70% increase in memory usage.

5.4.3 Moving an Object

When an object has been moved by the application, the octree has to be updated accordingly. The naive way to do this is first to remove all references to this object

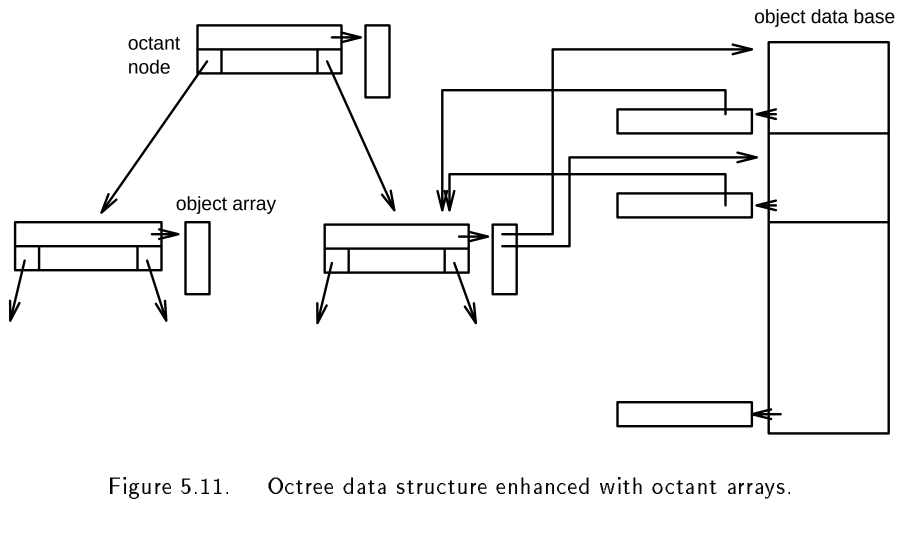


Figure 5.11. Octree data structure enhanced with octant arrays.

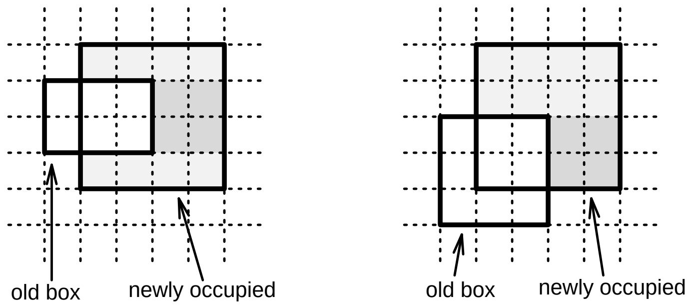


Figure 5.12. Symmetric set difference of "new" and "old" bounding boxes. The "new" part is partitioned into rectangular boxes. When moving an object, an octree traversal has to be done for only those cells which are in the box-difference.

from any octree cell, then to insert it again. This could be done by two traversals through the octree. With octant arrays used for neighbor-finding (see Section 5.4.2 and Figure 5.11), we can remove object references faster, but the insertion phase remains.

In most applications, however, objects move only a small distance (and rotate probably a little bit). This implies that their new bounding box is "almost" the same as of one frame before (*time coherence*). With two traversals for updating the octree, most of the references to an object would be removed only to be inserted immediately again with the next traversal.

Box difference. We cannot speed up the object removal phase very much (see below "Overkill"); all we can do is to scan the object's octant array and remove the object from all those cells which are no longer in the new bounding box.

Now we only have to insert the object in those cells which have not been occupied before. Given the "new" box $\boldsymbol{b}^{\text{new}}$ and the "old" box $\boldsymbol{b}^{\text{old}}$, we partition the difference $\boldsymbol{b}^{\text{new}} \backslash \boldsymbol{b}^{\text{old}}$ into (at most) six boxes (we will call them *entering boxes*):

$$
\begin{aligned}
E_0 &= [\boldsymbol{b}^{\text{new}}_{\text{xlow}}, \boldsymbol{b}^{\text{old}}_{\text{xlow}} - 1] \times [\boldsymbol{b}^{\text{new}}_{\text{ylow}}, \boldsymbol{b}^{\text{new}}_{\text{yhigh}}] \times [\boldsymbol{b}^{\text{new}}_{\text{zlow}}, \boldsymbol{b}^{\text{new}}_{\text{zhigh}}] \\
E_1 &= [\max(\boldsymbol{b}^{\text{old}}_{\text{xlow}}, \boldsymbol{b}^{\text{new}}_{\text{xlow}}), \boldsymbol{b}^{\text{new}}_{\text{high}}] \times [\boldsymbol{b}^{\text{old}}_{\text{yhigh}} + 1, \boldsymbol{b}^{\text{new}}_{\text{yhigh}}] \times [\boldsymbol{b}^{\text{new}}_{\text{zlow}}, \boldsymbol{b}^{\text{new}}_{\text{zhigh}}] \\
E_2 &= [\max(\boldsymbol{b}^{\text{old}}_{\text{xlow}}, \boldsymbol{b}^{\text{new}}_{\text{xlow}}), \boldsymbol{b}^{\text{new}}_{\text{high}}] \times [\boldsymbol{b}^{\text{new}}_{\text{ylow}}, \boldsymbol{b}^{\text{old}}_{\text{ylow}} - 1] \times [\boldsymbol{b}^{\text{new}}_{\text{zlow}}, \boldsymbol{b}^{\text{new}}_{\text{zhigh}}] \\
E_3 &= [\max(\boldsymbol{b}^{\text{old}}_{\text{xlow}}, \boldsymbol{b}^{\text{new}}_{\text{xlow}}), \boldsymbol{b}^{\text{new}}_{\text{high}}] \times [\max(\boldsymbol{b}^{\text{old}}_{\text{ylow}}, \boldsymbol{b}^{\text{new}}_{\text{ylow}}), \min(\boldsymbol{b}^{\text{old}}_{\text{yhigh}}, \boldsymbol{b}^{\text{new}}_{\text{yhigh}})] \times \\
&\quad\; [\boldsymbol{b}^{\text{new}}_{\text{zlow}}, \boldsymbol{b}^{\text{old}}_{\text{zlow}} - 1] \\
E_4 &= [\max(\boldsymbol{b}^{\text{old}}_{\text{xlow}}, \boldsymbol{b}^{\text{new}}_{\text{xlow}}), \boldsymbol{b}^{\text{new}}_{\text{high}}] \times [\max(\boldsymbol{b}^{\text{old}}_{\text{ylow}}, \boldsymbol{b}^{\text{new}}_{\text{ylow}}), \min(\boldsymbol{b}^{\text{old}}_{\text{yhigh}}, \boldsymbol{b}^{\text{new}}_{\text{yhigh}})] \times \\
&\quad\; [\boldsymbol{b}^{\text{old}}_{\text{zhigh}} + 1, \boldsymbol{b}^{\text{new}}_{\text{zhigh}}] \\
E_5 &= [\boldsymbol{b}^{\text{old}}_{\text{xhigh}} + 1, \boldsymbol{b}^{\text{new}}_{\text{xhigh}}] \times [\max(\boldsymbol{b}^{\text{old}}_{\text{ylow}}, \boldsymbol{b}^{\text{new}}_{\text{ylow}}), \min(\boldsymbol{b}^{\text{old}}_{\text{yhigh}}, \boldsymbol{b}^{\text{new}}_{\text{yhigh}})] \times \\
&\quad\; [\max(\boldsymbol{b}^{\text{old}}_{\text{zlow}}, \boldsymbol{b}^{\text{new}}_{\text{zlow}}), \min(\boldsymbol{b}^{\text{old}}_{\text{zhigh}}, \boldsymbol{b}^{\text{new}}_{\text{zhigh}})]
\end{aligned}
$$

Of course, these calculations are done using integer coordinates; that's okay, because min/max and +/− are commutative. Most of the time, 3 of the entering-boxes E_i are empty; all six are non-empty only if the box has grown but little moved.

Then we do an insertion traversal with each non-empty of the entering boxes E_i (see Figure 5.12). This should result in much fewer cells being visited.

5.4.4 Results

Some experiments were realized by which I tried to find out where octree algorithms could be improved.

Memory usage. The first test was designed to measure the pointer distribution with respect to octree depth. The octree comprised a 10^3-cube, containing 100 objects, each of size 1^3 (in local coordinate system). Each figure is an average of 1000 frames.

octree	#object pointers at depth (%)				
depth	1	2	3	4	5
2	0	100			
3	0	0	100		
4	0	0	0.6	99.4	
5	0	0	0.03	3.5	96.4

This means that with an octree of depth 5, say, 96% of all object pointers of the whole octree are stored in cells on level 5. (A depth 3 octree has got $8 \times 8 \times 8$ voxels.) It seems that an octree is an efficient space subdivision only if objects are large in comparison to the voxel size.

The same set-up was used to measure memory usage and actions. The octant and object arrays were implemented as growing/shrinking ones with hysteresis (see Section 7.3.4).

octree	usage	#mem. actions/frame		
depth	(kBytes)	malloc	realloc	free
2	8	9.4	78.7	9.4
3	16	113.7	78.2	113.7
4	264	839	55	839
5	2264	4821	14	4818

Memory usage includes the octree backbone plus all object arrays (not octant arrays). Variance seems to be very small (± 4 kBytes).

Timing. Figure 5.13 and Figure 5.14 show the results of some of the improvements over the naive algorithm, proposed in this section. Obviously, the algorithm which visits only those nodes it really has to is the fastest (as expected). All timings were done without rendering, only object movement, octree updating, and neighbor-finding (as a potential global stage of the overall collision module). The hardware was an SGI Onyx (4× R4400, 150 MHz).

Several other things have been tried to speed up the octree stage: a supposedly faster `malloc` routine was used; object arrays were not `free`d when they were empty. All of them did not yield any significant speed-up, if at all.

Performance of even the best algorithm presented here still seems to be rather slow (intuitively); this is odd, in comparison to the great success of octrees with ray-tracers.

The extensive performance measurements carried out suggest that that octrees (and maybe grids, too) are not the most appropriate data structure for solving the $O(n^2)$ problem on the global level. There is very little literature so far on space indexing data structures for highly dynamic environments. Octrees and grids

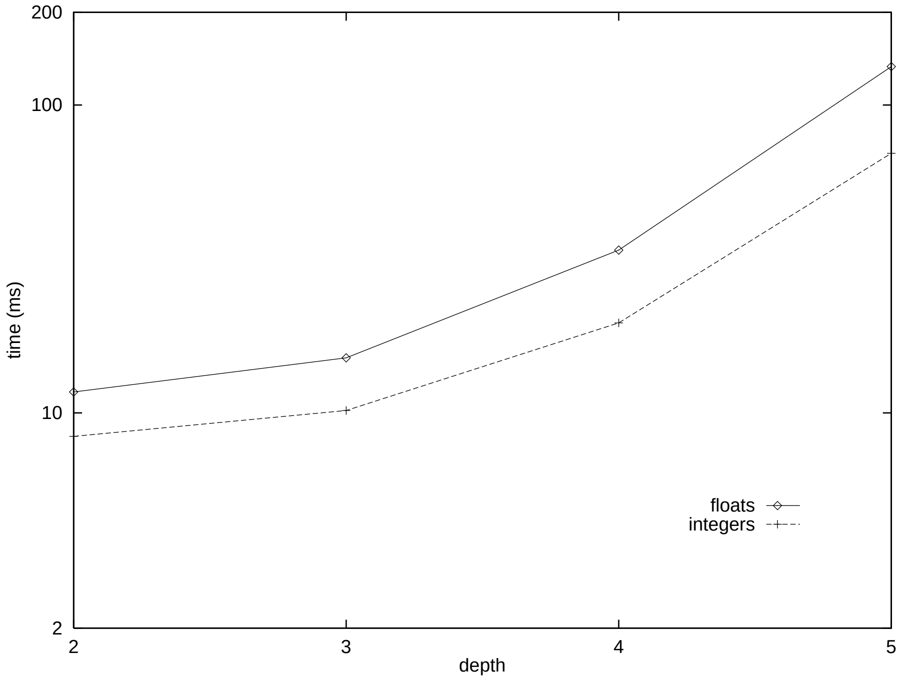


Figure 5.13. Both timings use the naive algorithm. A depth of 0 corresponds to the root only. (Unfortunately, the test program was not compiled with -O, so results here are not directly comparable to the ones of Figure 5.14.)

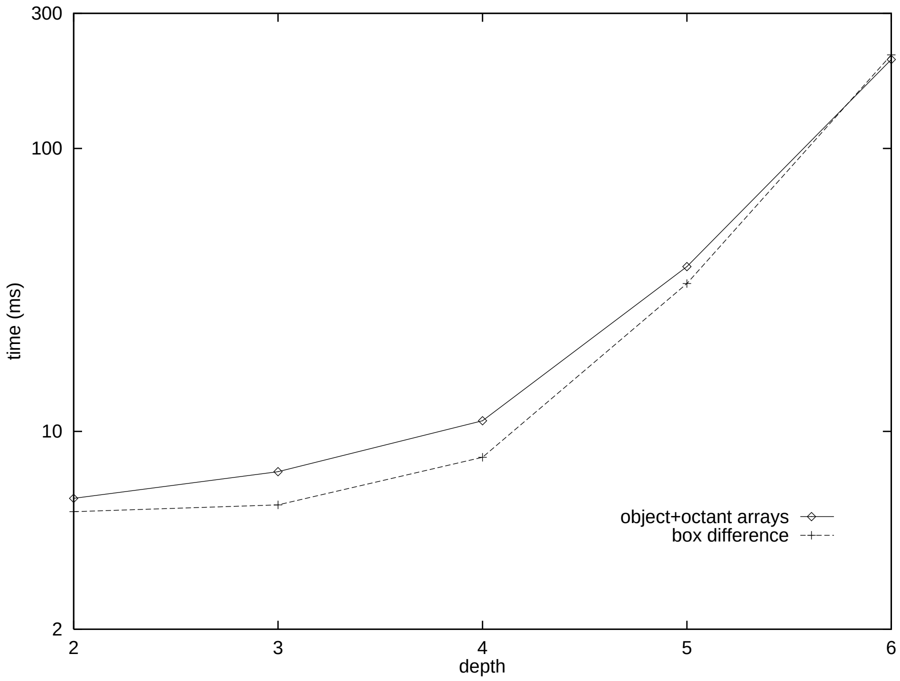


Figure 5.14. The first one uses octant arrays. The second algorithm uses the further optimized algorithm which visits only cells which are in the box difference. (Here, the test program was compiled with optimizer.)

have been used haevily in the area of ray-tracing and radiosity; however, these applications are of very static a nature (currently). So, for dynamic environments, other algorithms and data structures should be investigated.

5.4.5 Future directions

There are a few options, which could be tried further.

Pondering the first test in every recursion (*octant completely inside box*), one can see that all tests to decide it have already been made in (probably different) levels above the current recursion. These comparisons should be re-usable. (I didn't succeed in finding a coding which could decide the octant-in-box test without any further comparisons.)

Another idea ([Dai94]) is the following: when an object is inserted into the octree, we "attach" it to all those octants which are visited during the traversal (these are all octants where it is attached to with the current algorithm, plus all octants on the path towards the root). When moving an object, we now don't have to compare boxes, but instead can simply follow those paths which have the object attached to them. The disadvantage with this approach is: all objects are stored in exponentially many nodes (the estimation of [Dyer82] is no longer valid). Also, in order to find out if an object is attached to an octant node, it has to be searched for in the object array, which might take some time.

5.5 Grids

Most basic algorithms for grids are so simple, they hardly need to be mentioned. The general idea with grids is extremely simple: every cell which is (partially) occupied by an object will get a pointer to this object. A cell can, of course, hold many pointers.

The data structure of the grid is very simple; every cell has a list of objects attached to it, like with octrees. Objects do not need a cell array, which would contain pointers to all those cells which an object occupies, because we can enumerate those cells equally quickly by using the object's bounding box (see Figure 5.15).

When trying to find neighbors of an object, we just enumerate all cells by looping over the integer coordinates of its bounding box, and by looping over all object arrays of those cells.

When moving an object in the grid, we can use the same box-difference technique of Section 5.4.3 to change only those cells which have to be really altered.

Overkill. There is only one part of the algorithm left which looks at too many data items: the removal phase. When removing P, it does have to scan every object array of all those octant nodes where P is no longer contained in, in order to remove the pointer to P.

The idea was to add an index to every cell reference which tells where in the cell's object array the reference to P is stored (see Figure 5.16). This allows to remove any reference to P from the grid without searching any object arrays. At first glance, the advantage is offset just a little bit, because whenever we remove an entry in a cell array or in an object array, we have to adjust all indices into these arrays which point behind the entry being removed. This can be avoided, though, by moving the last entry to the place of the entry being removed (thus overwriting it).

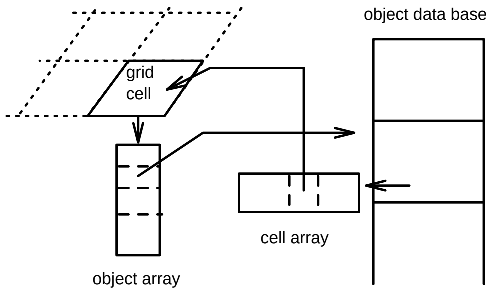


Figure 5.15. The simple data structure of grids.

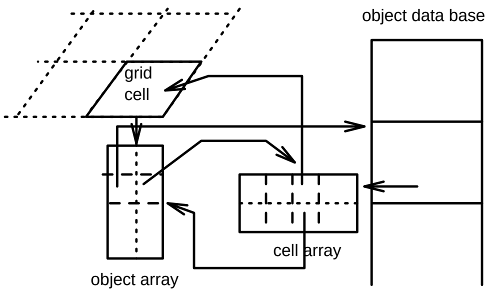


Figure 5.16. The grid data structure further augmented to allow for (supposedly) quick removal of object references from grid cells.

Unfortunately, this data structure performed worse than the simpler one (see Section 5.5.2)!

5.5.1 Non-axis-aligned Bounding Boxes

In the context of collision detection, it might be more efficient to add objects to cells according to their non-axis-aligned bounding box, even though the time spent with updating the grid will be much more. If axis-aligned bounding boxes in world coordinates are computed from the bounding box given in object coordinates, then an object will be attached to many more cells than are really occupied (see Figure 5.17). Using the tight bounding box, the number of neighbors in the grid might be much less and thus the number of object pairs which are tested for exact collision. So, altogether the overall time needed for a global collision check might be less.

Several methods were considered to do insertion of a non-axis-aligned bounding box into a grid.

The first one is to loop over all cells which are inside the aligned bounding box and check with Cyrus-Beck for intersection. Similarly to the box-tree algorithm, we can re-use many terms from previous iterations and from previous expressions. A little complication arises here, because we also have to deal with the case that a cell is completely inside the bounding box, or that the bounding box is completely inside a cell.

Another idea is to "chop" the box along the z-axis into *drums*, each one cell wide along the z-axis. These drums can have 3-6 vertices. Then we convert these drums to cell coverings. At first glance, one might think that this 2D problem is similar to the scan conversion of polygons — but it is not: with scan conversion, we are looking for the best approximation on a grid, here we want a covering.

As with scan line conversion, we can have very nasty problems, like many vertices falling into one z-slice (scan line), or very small angles at vertices.

In any case, this approach seems to be too expensive.

Stabbing the grid. The algorithm finally chosen is the *stabbing* idea: we shoot a ray along the z-axis through every integer vertex i, j on the xy-plane; considering all 4 rays at the corners of every "z-column" of cells (i.e., $1 \times 1 \times n$-box), we can enclose the box by a collection of "z-sticks" (i.e., $1 \times 1 \times k$-boxes). Once we've got

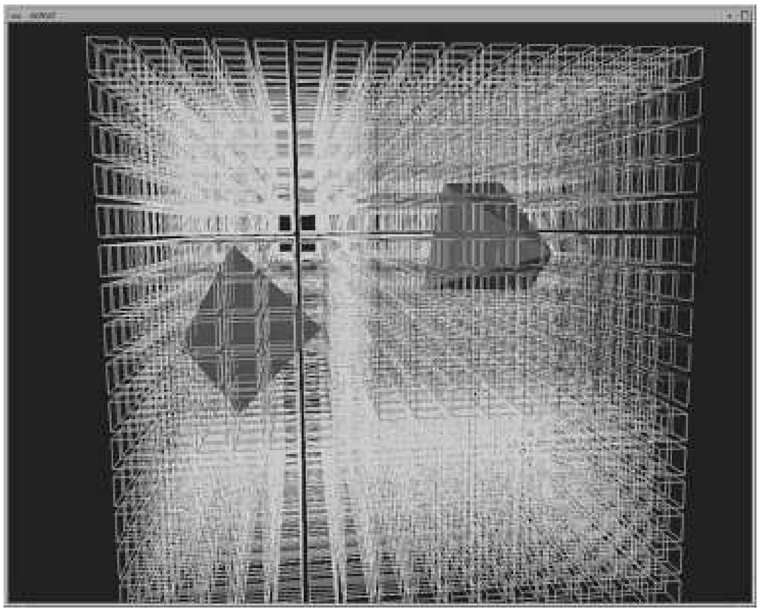


Figure 5.17. An object might be "attached" to many more cells of a grid than are actually occupied, if its axis-aligned bounding box is used. This test program colors cells which are occupied by any object, un-occupied ones are white.

those z-sticks, we can loop over all of them and add the object to all cells covered by them (see Figure 5.18).

We will stab the grid only by rays which pass through the axis-aligned box; also, stabbing it at the border of the axis-aligned box is not needed, since those rays can penetrate the non-axis-aligned box if it is actually parallel to the z-axis.

In the description of this algorithm we'll assume that we use z-rays; in order to minimize the number of stabbing rays, an implementation will choose that side of the axis-aligned box which is smallest.

Again, a box B is given by a point p and three spanning vectors b^x, b^y, b^z. The six faces of the box will be called x-, y-, and z-planes. A ray through grid coordinates i, j will hit, say, the two x-planes of B at line parameter t^x_{ij} and $'t^x_{ij}$,

$$\begin{aligned} t^x_{ij} &= \frac{b^x(p-q)}{b^x_z d_z} - i\frac{b^x_x \Delta_x}{b^x_z d_z} - j\frac{b^x_y \Delta_y}{b^x_z d_z} \\ 't^x_{ij} &= t^x_{ij} + \frac{1}{b^x_z d_z} \end{aligned}$$

As we can see, when varying i and j, all t^x_{ij} can be calculated by two float additions, and $'t^x_{ij}$ by just one more. All other terms can be pre-calculated.

Analogously, we compute the t^y- and t^z values for the other two slabs of the box; this can be done in one loop. After having computed all t-values, we compute the line parameter intervals T_{ij} of that part of each z-ray which is inside the box B.

Then, we compute the z-range of all z-sticks by just taking the min and max of the four T-ranges $T_{ij}, T_{i+1,j}, T_{i,j+1}, T_{i+1,j+1}$ (depending on which side the rays enter/leave).

Again, because we know several things about the geometry: for example, z-rays always go from $-$ towards $+$, entering/leaving status stays the same over all z-rays

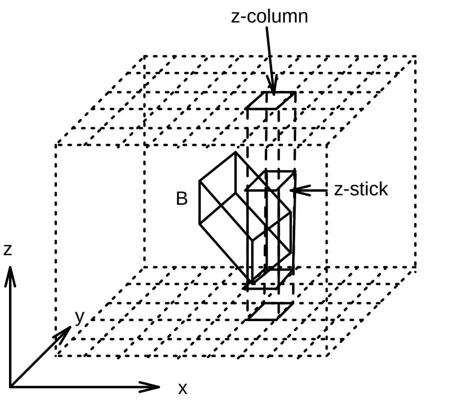


Figure 5.18. Enter a non-axis-aligned box by stabbing the box at integer coordinates and enclosing it in $1 \times 1 \times x$ axis-aligned boxes.

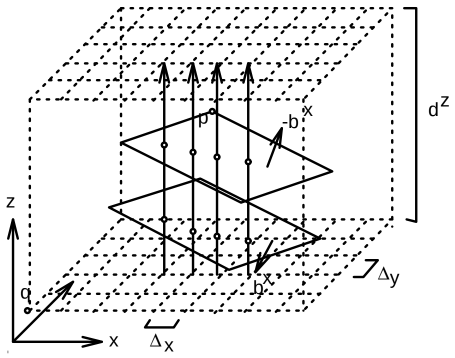


Figure 5.19. Most of the terms from a previous iteration can be re-used.

for a certain plane pair, $T^{\min} > T^{\max}$ cannot happen, etc. — because of that we can optimize the code considerably.

As with the box-tree algorithm, we have to take care of the special case that some (at most 4) planes of the box are parallel to the z-axis.

Actually, this method might miss a cell which is occupied by the box, if none of the z-rays through its corners penetrate the box. This might become a problem if the boxes are very small in comparison to the cell size. To reduce this effect, we additionally occupy all those cells which contain one of the 8 box vertices.

5.5.2 Results

The first test was carried out to compare the "simple" algorithm with the "improved" algorithm; by "simple" I mean the one using object and cell arrays, which also uses the box-difference technique of Section 5.4.3; by "improved" I mean the one which uses the doubly-pointered object and cell arrays which mutually contain also indices into the other array. Results can be seen in Figure 5.20.

The test was done, as usual, without any rendering; it did take into account transformation matrix, bounding box calculations, grid updates, and generation of all object pairs which would have been handed to some exact collision detection algorithm. The scenario was the usual cage of size 10^3 with 100 objects of size 1^3 flying around; each figure is the average on 1000 frames.

A big shock was the result of the "improved" data structure described in "Overkill" (which maintained array indices, too). With a grid size of 32^3, this data structure performed by a factor of *1.5 slower* than the algorithm which searched for object pointers in the cells. The object arrays stored at each cell are quite small – still, this cannot explain why the augmented data structure is actually slower! I could only come up with the following explanation: with every pointer de-referencing, a part of the memory is addressed which is far away from where the pointer was stored. There are 8 pointer de-referencings per removal from one cell. My guess is that each de-referencing caused a cache miss, which would cause the CPU to swap the cache in and out 8 times per loop iteration.

Comparing memory requirements is as frustrating as comparing their time characteristics:

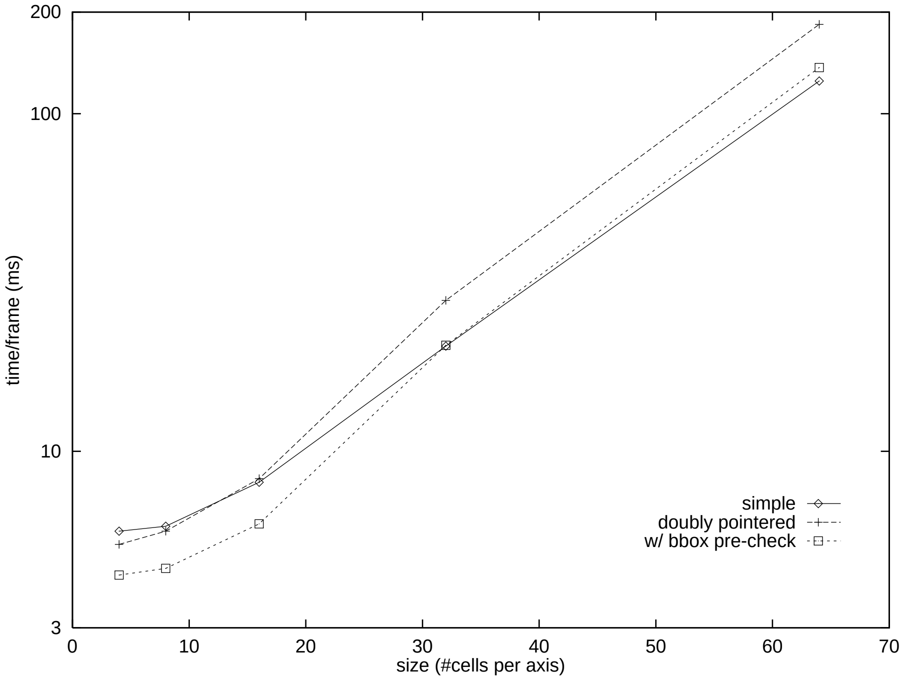


Figure 5.20. Comparison of the "simple" and the doubly-pointered grid data structure. (A size of n corresponds to an octree's depth of $\log_2 n$.)

grid size	memory (kBytes) simple	w/indices
4	16	24
8	16	40
16	168	744
32	1,808	6232
64	14,752	–

The third curve in Figure 5.20 is the simple algorithm with the additional pre-check whether the new and the old bounding boxes (in integer coordinates) are the same (if so, the function is finished).

Figure 5.21 shows a comparison of the simple algorithm (including bounding box pre-check) and the one which takes non-axis-aligned bounding boxes.

Finally, some tests were made with exact collision detection after the grid phase. The goal was to find out at what scene complexity a grid pre-stage actually payed off.

The scenario of Figure 5.22: a certain number of tetrahedra (bbox-size 1^3) moving in a cage of size 5^3; each figure is the average of 5000 frames[2]. Obviously, even $100^2/2$ bounding box checks are very cheap compared to some superfluous exact collision detections. Hardware: 150 MHz R4400.

Another scenario with exact collision detection is: 30 moving objects (spheres, tori, cube arrays, tetra-flakes, plus the 6 wall boxes) in a cage of size 5^3, each with 50–750 polygons, together 4758 polygons[3]. Rendering was switched off, each figure

[2] Invocation: `movem -m` *num-tetras* `-s g` *size* `-a cx -e 5 -t 5000 sh`.

[3] Invocation: `movem -m +30 -x 7 -s g 8 -a bx -e 5 -t 5000 sh`

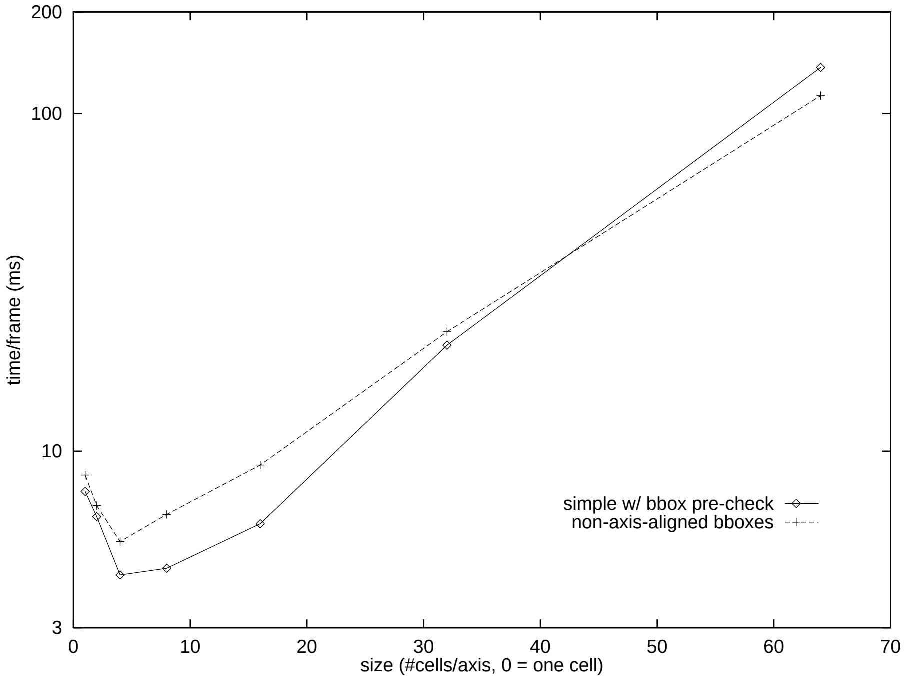


Figure 5.21. Comparison of grid space subdivision, using axis-aligned boxes and non-axis-aligned boxes. Grid-phase only (no exact collision detection afterwards).

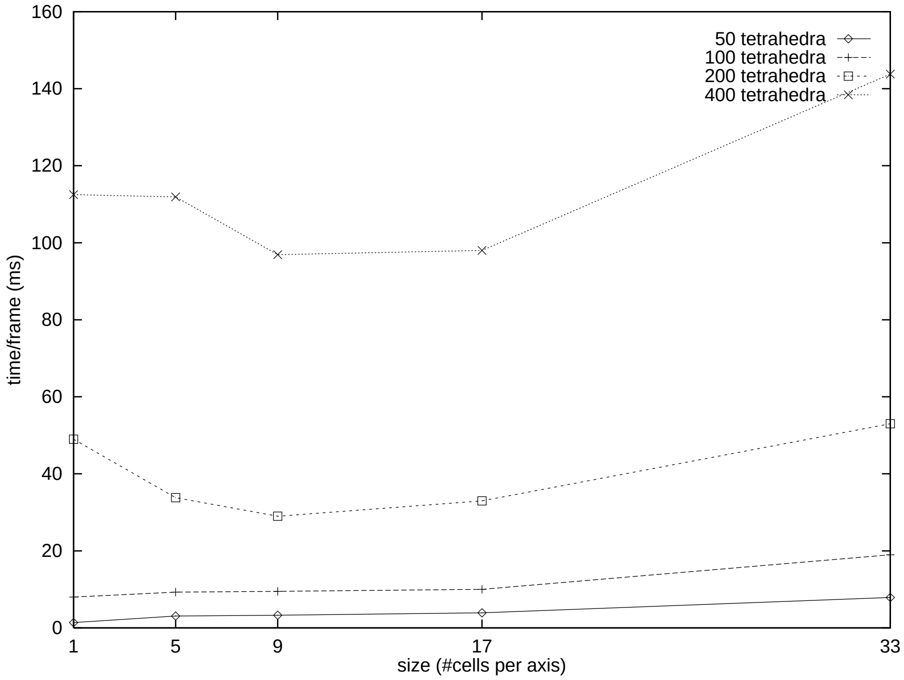
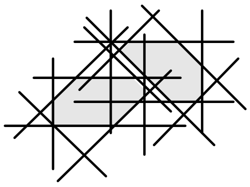
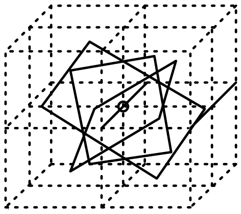


Figure 5.22. The grid pre-phase combined with exact collision detection among tetrahedra.



(a) Bounding volumes made of more than 3 slabs (here 2D)



(b) Corresponding octree

Figure 5.23. Generalization from the conventional 3-slab bounding volumes.

is the average of 5000 frames. The test was run on SGI with a 40 MHz R3000 (VGX):

method	time/frame (msec)	# obj.-pairs after grid
no grid, no box-trees	260	615
no grid, with box-trees	234	615
with grid, with box-trees	219	155

The last one considered the special case where only one object moves among many stationary objects. The scenario here was: one moving object (20 polygons) among 500 fixed polyhedra (tetra-flakes), each with 52 polygons, altogether 25,000 polygons[4]. Rendering was switched off, exact collision detection was done:

without grid	1.0 msec/frame
with 8^3 grid	0.2 msec/frame

5.6 Generalized Octrees

What is the connection between space-subdivision schemes and bounding volume types? With conventional octrees the correspondence is as follows: given two cubes in world coordinates by their enclosing slabs

$$s_i \ : \ n_i x - d_i < 0 \ , \ -n_i x - d_i' < 0, \qquad i \in 1, 2, 3$$

we can check for overlap by simply comparing their d_i's and d_i's. We can do that with n slabs, too [KK86] ($n \geq 3$; see Figure 5.23(a)).

Correspondingly, an octree is recursively constructed by taking a large bounding volume which encloses the whole "world". This volume is then divided at its center by all n planes which slabs are made of (see Figure 5.23(b)). When inserting an object, we recursively compare corresponding d-values of the object's slabs and the cell's dividing slab-planes.

A little problem seems to arise here: I suspect that an arbitrary bounding volume made of slabs will not yield similar bounding volumes when divided by slab planes at its center. But I do *not* think that that would be a problem; we could just use the center of the conventional octree's cubes.

This method has not been implemented, yet. It seems to me that it offers advantages in comparison to conventional octrees, because the depth can be much less while still achieving the same "resolution"; more important, I think that this "octree" scheme combines naturally with the generalized slab bounding volumes, which are on average much tighter than boxes or spheres can be!

[4] Invocation: `movit -m 500 sh -t 10000 sh -e 20 -x 3 -s n 8 -a ar`.

Generalized grids? It is not clear to me whether this scheme would also work for uniform grids. The problem I see there is: given a slab bounding volume, how could we enumerate all cells of the slab grid which are occupied by the volume?

5.7 Without Space Subdivisions

There are other methods to solve the all-pairs problem apart from space subdividing. They have not (yet) been implemented, though.

Bounding box hierarchies. These try to exploit the fact that users tend to model hierarchical scenes by grouping those objects in the same sub-tree which are geometrically close [YW93]. However, if the scene is highly dynamic, this relation could get spoiled very soon, i.e., the bounding boxes of assemblies could get very large, because their children have moved far apart.

For ray-tracing they seem to work very well [KK86], because the scene is entirely static; so, the hierarchy can be built once at start-up time. In dynamic environments, the hierarchy would have to be rebuilt every n-th frame in order to maintain a fairly optimal correlation between spatial vicinity and vicinity within the tree. Since ray-tracers have to build the hierarchy only at start-up time, they can create a much finer and better optimized hierarchy on polygon level.

Another problem is the following: In a given application, there might be many objects which we are not interested in any collision detection at all (see Section 7.0.1). When looking for collisions with a given object A, there is no way to avoid looking at those other objects. We might keep a counter at every assembly node whether there are any "collidable" objects underneath it – but that would not help in general, since there might be just one object at some leaf way down in a deep sub-tree.

If the hierarchy is very flat (i.e., just the root and almost all other nodes are leaves, which can happen if the data originate from CAD systems), then we will be back to the n^2-case!

Sorting. Here, we're given just a collection of axis-aligned boxes which might change size and move around.

The basic idea is to *sweep* a plane along the x-axis (see Figure 5.24); whenever this plane intersects more than two boxes at a time, we check all these rectangles, obtained by the actual intersection of the sweep plane at the current position with those boxes [Edelsbrunner94, BF79, PS85]. This is now a similar test in two dimensions, which can be performed by the same method, now with a sweep line instead of a plane.

Since there are only finitely many boxes, we can first *sort* the set of x-values of all boxes (x_{min} and x_{max}). Then, the sweep of a plane along the x-axis can be done by "hopping" from one x-value to the next in the sorted list. During the sweep, we maintain a list of y-values and z-values. At each x-value found in the list we either *enter*, or *leave* a box; correspondingly, we add or remove its y- and z-values from the y- and z-value arrays. Since the sweep along the y-plane will need the y-array to be sorted, too, we will add and remove y- and z-values by insertion sort (which involves at most one pass over the two lists).

This method is very suitable for exploiting *time coherence*: since objects won't move very far between two frames, the list of x-values of a certain frame will be "almost sorted" with the next frame, too. This suggests to keep this array and update it by a few bubble-sort passes.

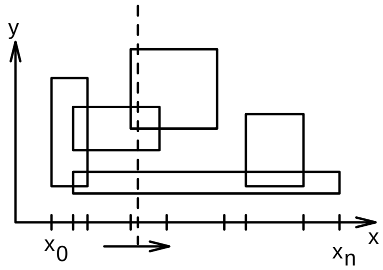


Figure 5.24. The sweep-plane algorithm considers only x-coordinates where the intersection/non-intersection status of any two boxes might change.

In my opinion, this seems to be the most promising algorithm to solve the all-pairs problem on the global level. However, there have been practically no results on octree/grid performance for dynamic environments (as yet). The limited time frame for this thesis did not allow to investigate this direction any further.

Also, the CPU time spent by one of the methods presented above is neglectable compared to any pairwise collision detection algorithm.

Chapter 6

Parallelization

With today's trend towards multi-processor architectures, any problem should be considered for parallelization. In particular, the collision detection problem seems to be easily and efficiently parallelizable.

There are basically *three levels* to characterize parallelizations:

- *concurrent*,
- *coarse grain*,
- *fine grain*

and *two load balancing schemes*:

- *static work load* balancing; every process is assigned a certain portion of the total work at start-up time; when it is finished, the process waits for the others to finish, too.
- *dynamic work load* balancing; every process retrieves a small portion of the total work; when it is finished, the process tries to retrieve another portion until there is nothing left.

The classification is somewhat fuzzy, i.e., there are no clear border lines.

Dynamic work load balancing burdens each process with a little bit more synchronization overhead, but it is highly recommended. So, I decided to implement it in all parallelized algorithms; even more so, since the overhead does not seem to be very high in the context of collision detection.

There are many possibilities to distribute the algorithms presented so far on several processors, especially because on each of the levels mentioned above we have almost complete *data independence*:

- There are at least two different concurrent approaches:
  - The whole collision detection module runs concurrently to application and/or renderer*;
  - When testing an object pair for exact collision, we start different algorithms in parallel with the same pair for input*.

    We could choose three complementary algorithms, one which is fast when objects do not collide, the other which is fast when objects do collide, and a conventional one. Then, if either of the three has found a solution, all three of them terminate.

*items marked have been implemented

- On the global level (this corresponds to the coarse grain level):
  - Updating the spatial data structure (grid or octree) can be done by (conceptually) one processor per moved object.

    There is probably a substantial overhead with this approach: every access to a cell's object array must be exclusive, i.e., guarded by semaphores (or locks). An access collision will occur the more often the more processors are applied to update the grid/octree. We would need either one semaphore for every cell individually, or, if we use only one semaphore for any access to any cell's object array, the collision probability would be intolerably high.

    But the overall gain of parallel grid/octree updating would not be substantial, probably, because this task is a minor part of detecting all collisions in a highly dynamic environment.
  - If we assume that we are given a list of object pairs to be tested further for exact collision, we can process every pair in parallel without any further overhead, even if the same object is part of several pairs*.

    Of course, a real implementation has only a limited number of processors, so there will be a little overhead for guarding that region of the code which takes the next pair out of the list.
- On the edge-polygon level (this is the fine grain level):
  - The algorithms of Sections 3.2 and 3.4 can be distributed on two processors while still having very little overhead*: given an object pair (P, Q), one processor can test edges of P against Q, the other can test edges of Q against P.

    This is sort of a *medium grain* parallelization.
  - Testing a set of edges against a set of polygons (or clipping them against a convex volume) can be distributed on as many processors as there are edges. This is parallelization on (almost) the finest level possible.

    With less processors than edges, the overhead can probably become very high, because retrieving the next un-tested edge has to be exclusive, thus guarded by a semaphore. Since a single edge-polygon test is very quick, processors will spend a lot of their time waiting in the queue of the semaphore.

    This could be improved a little by retrieving not just single edges but chunks of un-tested edges. Static work balancing is probably not an option, because edges tend to be grouped by vicinity in the edge array, which means that most edge-polygon tests will be finished with the edge-bbox test, which is very soon.

6.0.1 Parallel Computation of Bounding Boxes and Transformation Matrices

Geometry nodes in the object hierarchy contain some information which is needed by all collision detection algorithms (like transformation matrices), but which, unfortunately, depends on information which is stored at other nodes (see Figure 6.1 for a depiction of the dependences).

Each node holds several local transformations and a matrix describing its transformation to world coordinates (this will be called its *to-world* transformation).

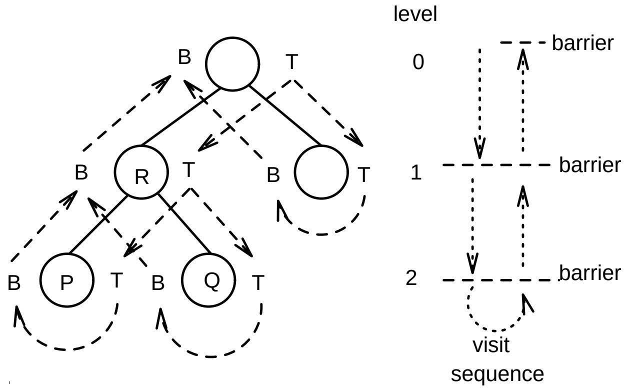


Figure 6.1. Some dependences of information in the object hierarchy. Visit sequence of multiple processes; between each two barriers, processes are data-independent.

When an application changes a local transformation of a node, its to-world matrix becomes invalid, *and* the to-world matrices of all its direct and indirect children become invalid. Whenever the application calls a function to calculate a to-world transformation matrix, the object handler has to make sure that the to-world matrix of the parent node is valid. This could eventually cause the object handler to calculate the root's transformation matrix.

Bounding boxes exhibit a similar dependence scheme, except that it is turned "upside down". Thus, when the object handler is requested to compute the root's bounding box, this could eventually lead to the calculation of all nodes' bounding boxes.

In the context of parallel collision detection, this data dependence looks like a serious problem at first glance, since several processes might compute the same transformation matrix or bounding box at the same time. A few things can be done about that:

- Calculate all bounding boxes and to-world transformation matrices sequentially before any collision detection algorithm will need them.

- First, calculate all to-world transformation matrices of all moved objects in parallel. This might cause two or more processes calculate the same transformation matrix at the same time; however, input and output of the responsible function are completely independent, so that two or more processes would just overwrite each other's results with the same data (see Section 6.2 for a full explanation).

  Second, calculate all moved objects' bounding boxes. Since all moved objects are leaves, no bounding box will be computed by more than one process.

  This is the approach taken here, for the time being.

- Use a parallel algorithm as depicted in Figure 6.1 (I will call it the *jo-jo method*): several processors compute on the same level; when they're all finished with that level, they all shift from one level to the next synchronously (barrier-synchr.).

  First, they would all proceed from top to bottom computing transformation matrices, then back up again, computing bounding boxes (just following the data dependences).

6.1 Terminology and Limits

When parallelizing algorithms, we usually want to know how they compare to their sequential counterparts. To that end, we need to define a few terms. Let T_1 be the time of a certain algorithm A on a 1-processor machine, let T_p be the time of a parallel algorithm B, solving the same problem as A, on a p-processor machine. Then we say that B gives a *speed-up* of

$$S_p := \frac{T_1}{T_p}$$

by using p processors. It is necessary to mention A and B, since we are talking of algorithms, not of problems. Of course, S_p has significance only when A and B are (in some sense) "good" algorithms !

The *efficiency*

$$E_p := \frac{S_p}{p} = \frac{T_1}{pT_p}$$

tries to provide a measure for the quality of the parallelization, of B. It is also sort of a measure whether it is worth parallelizing A, because it is clear that p processors can solve most problems faster than 1 processor can; on the other hand, p processors are, at least, p times as expensive as 1, so, to be really cost-effective, they should be p times as fast.

The goal is always to get $E_p \equiv 1$ — unfortunately, almost always $E_p \to 0$ for $p \to \infty$!

A few hypotheses. There are a few statements on what might be realistic when parallelizing code.

Minsky's Hypothesis says that

$$\forall\, p: \; S_p \leq 1 + \log p$$

which is an empirical observation. If this is true, then it does not make sense to use massively parallel systems, because $E_p \leq \frac{\log p}{p} \to 0$ for $p \to \infty$!

Amdahl's Thesis presumes that every algorithm has an inherently sequential part $\nu, 0 \leq \nu \leq 1$, which cannot be parallelized. Then this algorithm can have a maximal speed-up[1]

$$S_p \leq \frac{1}{\nu}$$

This would imply[2] that it would not make sense to increase $p \geq \frac{1}{\nu}$. However, this thesis is correct only if we take a sequential algorithm and parallelize that (for example, by an automatically vectorizing compiler); if we invent an entirely new algorithm specially suited for the parallel architecture, then the speed-up can be much better.

The *Law of Gustafson* seems to claim quite the opposite of Amdahl's thesis[3]:

$$S_p = \nu + (1 - \nu)p$$

The outcome of all this is just: Amdahl's law says that for a given problem we cannot reduce the computation time arbitrarily by just applying more processors to it, whereas Gustafson's law says, that parallelization can help tackle problem sizes which we otherwise would not be able to solve at all.

[1] $T_p = \nu T_1 + (1-\nu)\frac{T_1}{p} \;\Rightarrow\; S_p = \frac{T_1}{\nu T_1 + (1-\nu)\frac{T_1}{p}} = \frac{1}{\nu + \frac{1-\nu}{p}} \leq \frac{1}{\nu}$

[2] Since $S_p \leq p$, choose $p = S_p \;\Rightarrow\; p \leq \frac{1}{\nu}$

[3] this follows from the assumption that $T_1 = (1-\nu)pT_p + \nu T_p$

6.2 Detecting Multiple Collisions in Parallel

This section will deal with conventional coarse grain parallelization, i.e., the same function is executed on several processors at the same time to solve different instances of the same problem.

This algorithm tries to do as much as possible in parallel on the global object level, except for updating any space indexing data structure, because, as was said above, I suspect this would have too much synchronization overhead.

The basic idea is: given the list M of objects which have moved since the last frame, take objects out of this list in parallel and update their bounding box and to-world matrix. If all are updated, update the grid/octree sequentially. Then, find sequentially all object pairs which are "close" to each other and store these pairs in a list P. After that, process the list P in parallel.

Pseudo-code for this scheme follows:

<u>`Global parallelization`</u>

`exec parallel:`
    take next object o out of list M
    calculate to-world matrix and bbox of o
$\forall\, o \in M$: *update o in grid/octree*
$\forall\, o \in M$:
    put all objects pairs (o, o') in list P,
    if o and o' occupy a common cell,
    and if it is not yet there
`exec parallel:`
    take next object pair (o, o') out of list P
        check (o, o') for exact collision

Only those parts of the function which are marked by dots are exclusive regions. These regions are guarded by a lock, which keeps the process spinning (i.e., busy waiting), so as to minimize synchronization overhead.

As mentioned above, we are allowed to do the matrix and bounding box updating in parallel *without* guards, in this particular case. Let's assume that two processes are about to calculate the to-world matrices of object P and Q, resp. (see Figure 6.1). Then, both realize that they need the to-world matrix of the common father R. So they both start to calculate R's matrix at the same time, which results in multiple writes to the same memory locations.

However, this is not a problem. First, the bus architecture does not really allow concurrent writes — there are only exclusive writes, so results are not undefined with these conceptually concurrent writes. Second, the matrix calculation function is coded such that input data (i.e., the local transformations) and output data (i.e., the to-world matrix) are independent. And third, both processes write exactly the same data, so they do not destroy each other's results. If matrices further up the object hierarchy had to be computed, too, the same arguments would apply again.

This method is not the most efficient way to compute several transformation matrices and bounding boxes in parallel, especially if there are many objects and if the tree is deep. In that case, the "jo-jo" method might be more efficient (see Figure 6.1).

6.3 Parallelizing a Single Object-Pair Request

This section describes two methods how to parallelize a single request whether or not a pair of objects collide.

6.3.1 Fine-Grain Parallelization

As mentioned above, the arbitrary and convex algorithm can be parallelized on an edge-polygon level, because each edge-polygon test is entirely independent of all other tests. The whole collision check can be distributed on many processes by assigning each of them a certain portion of the edge array (or even a portion of the polygon array, too). For dynamic work-load balancing, each process can take out a portion of the arrays itself.

We can parallelize even more: the calculation of world coordinates of vertices and normals can be distributed in a similar fashion. Between each stage (vertices, normals, edge-polygon tests), we could use barrier synchronization. We can probably even do without such a synchronization: when a process cannot take out another portion of, say, the normal array, it starts to take out the first portion of the edge array and check them or clip them.

The box-tree algorithm could be parallelized in two ways: One could do the simultaneous box-tree traversal with one process and collect all intersecting elementary boxes in an array. Then this array could be processed in parallel. Or, we could try to parallelize even the box-tree traversal; however, this would imply that a process is spawned with every recursion, which might be very expensive.

The separating planes (see Section 3.4.2) algorithm can be parallelized, too; however, the synchronization overhead might be higher than that of all other algorithms. Every plane w_i (which is checked whether or not it is a separating plane) depends on w_{i-1} of the iteration before and some point z_{j_i}, i.e., $w_i = f(w_{i-1}, z_{j_i})$. This is the worst data dependence one can have — usually, it means that the algorithm is not parallelizable. In this particular case, we can distribute the check whether or not w_i is a separating plane (barrier synchronization). Experience seems to indicate that barrier synchronization usually incurs a very bad overhead.

The probabilistic algorithm for closed polyhedra (see Section 3.5) is as easy to parallelize as the arbitrary or convex algorithms. After all normals have been transformed (maybe in parallel, too), each process tests random points for "inside" in both of the polyhedra. If one of them finds a point to be inside, all processes are finished.

6.3.2 Medium Grain

If there's only one object pair to be tested for a certain frame (which could happen fairly often if there's only one moving object), then we can use *two* processors to parallelize collision detection.

If we had very many processors, we could combine medium grain with coarse grain parallelization.

The algorithms presented in Sections 3.2, 3.4, 3.6 consist of two phases: check edges of P against polygons of Q, and edges of Q against polygons of P. The worst case with these algorithms is an "almost" collision; in that case, both phases have to be run, and many edges and polygons have to be tested.

The idea is simply to run the two phases of the algorithms in parallel. If one of the processes finds a collision, it sets a flag so that the other process terminates, too; if neither of them finds a collision, then the overall run time will be about half of the sequential time.

There's also the case that one of the processes finds an early "non-collision", which can also be signaled to the other process by the same flag.

The outline of the parallel code is

```
Two phases parallel

input:  polyhedra P, Q
result := "none"            {flag to tell other process if sth. found }
exec 2× parallel:           {process 0 is called with A = P, B = Q }
    collect polygons of B   {process 1 is called with A = Q, B = P }
    which are in A's bbox,
    there are none
     ⟶
         result := "no collision"
        ↑ result
    ∀ e ∈ A:
        result set   ⟶    return
         e intersects polygon of B
         ⟶
             result := "collision"
            ↑ result
```

Theoretically, we would have to guard the regions which set or test the flag `result`; however, since the bus prohibits concurrent writes, the worst thing that could happen is that the other process iterates the edge loop once too often.

6.4 Concurrent Collision Detection

The one expectation that a virtual environment necessarily has to meet is *immersion*. The most important human factor to achieve immersion is a constantly interactive frame-rate. To this end, all parts of a virtual reality system must be able to adapt to an increase of the work-load, i.e., they must provide some way to reduce their share of CPU time — maybe even at the expense of reduced accuracy.

One method to attain constantly interactive frame-rate is to decouple all those parts from the VR system which are not directly involved in feeding input data to the renderer.

Such a decoupling could be done by unsynchronized concurrent loops with some kind of communication in-between. The model chosen here is the classic *producer-consumer* model, maybe with the exception that the buffer in-between is not, as usually, a FIFO but just two swapped buffers. (A few other deviations are discussed below.)

Two reasons led to the decision not to use FIFOs:

- "Back"- and "front"-buffers are needed anyway (see Section 7.0.1).
- When a collision is detected, the collision detection module executes a callback to the application. This callback can change transformations of the colliding objects (in fact, all my test programs do so). This transformation change would make it necessary to enter the object immediately again into the buffer — which could cause never ending loops or the buffer to get clogged.

The advantage of a FIFO is that it does not need any lock. With two buffers we need to guard every write (not the reads, though) by a lock; but since a buffer swap occurs relatively rarely (see below), the application will very seldom have to wait when writing an object into the front-buffer.

Concurrency in the context of collision detection poses other potential problems which are non-trivial to solve:

- Before an object is tested for collision with other objects, its vertices and normals have to be transformed to world coordinates. After the matrix has been calculated, all vertices are transformed. For the time being, the collision detection module can't make sure that this matrix stays the same during transformation of all vertices.

  Theoretically, another process (even the application itself) might change the transformation *and* the to-world matrix while the collision detection module is still transforming vertices.

  However, this did not happen so far, because currently, there is no other process which changes transformations and needs the to-world matrix.

  This problem is not trivial to solve: the naive solution, which would just lock access to the transformation matrix of the object while the collision detection module is transforming vertices, does not seem to be satisfactory.

- The callback (which is a function within the application) cannot know as of which time (frame) the collision has occurred.

  What happens with my test programs regularly is that objects always move a little further before their collision is detected.

  In order to overcome this problem, the collision detection module would have to tag objects with some time information, or, better yet, it should keep the current transformation as of the time when an object is entered in the buffer for later use by the application.

The conventional producer-consumer model has to be modified a little to suit the needs of collision detection in virtual environments:

- Usually, the overall program (producer plus consumer) is not allowed to loose any data on its way from producer to consumer. This means that if the buffer in-between (FIFO or swapped buffers) is full, the producer has to wait.

  Not so in the applications intended to use collision detection: the producer must never be kept waiting. The only solution to that problem is to throw away objects which cannot be processed quickly enough.

- Every object is entered at most once in any buffer. (In contrast, usually each data item is entered in the buffer as it is generated by the producer.)

  Although an object is entered only once in the buffer, its transformation can still be changed, even after front- and back-buffer have been swapped.

- The collision detection module has to make two passes over all objects which have been moved since the last frame.

- In a conventional implementation using two buffers, when the consumer has emptied the back-buffer it waits for the producer to fill the front-buffer — then

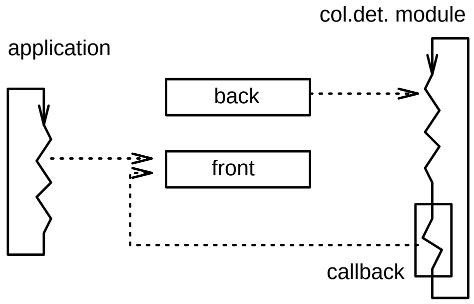


Figure 6.2. Flow of data and control of concurrent collision detection and application.

the buffers are swapped. This technique allows to implement the swapping operation safely without locks.

But, in the current context, it does not make sense for the collision detection module to wait until the application has filled the front-buffer — we want any collision to be detected as soon as possible. So, whenever the collision detection module is finished with the back-buffer, they are swapped.

- The distinction between producer and consumer is somewhat arbitrary here, because the consumer (the collision detection module) also gives some input back to the application, i.e., collision detection events. Of course, these have been requested by the application.

Because of the deviations mentioned above, the swap of the two buffers and the entering of an object in the front buffer have to be exclusive. First, I tried to get away without a lock, since the swap operation itself takes only about 3 assignments. However, the"very rare" case actually happened, so I had to add the lock. The overhead is barely measurable, anyway, because a swap does not happen too often (less than once per frame).

Figure 6.2 depicts the flow of data and control; the following is pseudo-code for the concurrent collision detection loop:

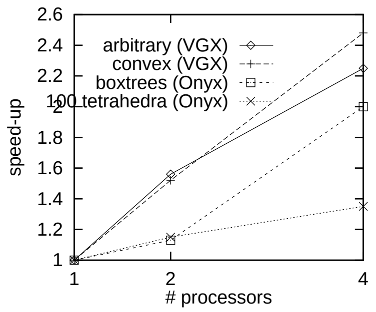
<u>`Concurrent collision detection`</u>

`loop forever:`
    front buffer empty $\longrightarrow$
        `sleep`
    something in front buffer $\longrightarrow$
        swap front- and back-buffer
        process back-buffer

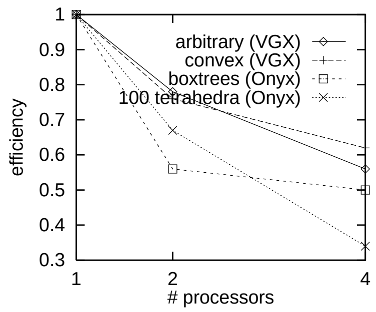
6.5 Dual Concurrent Algorithms

The basic idea here is to exploit certain algorithm "behavior": some algorithms find, on average(!), a collision rather early, some find a non-collision relatively early, compared to their worst-case, resp. For example, both, the arbitrary and the convex collision algorithms, will detect a collision earlier (on average) than a non-collision. On the other hand, the separating planes algorithm will detect a non-collision earlier than a collision.

The idea is to tie these two together and thus get a better overall performance. For implementation reasons (and in order to keep the sequential code fast), the



(a) Speed-up



(b) Efficiency

Figure 6.3. Performance of global parallelization.

original functions have to be duplicated and slightly modified. The difference, in the edge/face loops, is that each function has to "listen" if the other function has found a result, and if so, terminate; additionally, each function has to "tell" the other one if it has found a result.

An implementation was done for the arbitrary and the box-tree algorithm together with the separating planes.

In Section 3.4.2 we introduced an incremental version of the separating planes algorithm. It keeps the plane for every object pair which has been calculated the last time. This method can be still used here; it is perfectly compatible with the idea of dual concurrent algorithms. (The implementation does keep and update these planes.)

6.6 Results

Coarse grain. The first tests were made to find out the efficiency of the global parallelization (see Figure 6.3). Tests were carried out on various architectures and with various pairwise algorithms[4]. In any case, only up to four processors were available. The VGX is an SGI Skywriter VGX with four R3000 processors (40 MHz), the Onyx has got four R4400 processors (150 MHz). Rendering was switched off.

The speed-up for the extreme case with 100 tetrahedra is smallest. I suppose, the reason for this is that the collision check for two tetrahedra is very quickly done; the time spent with collision checking (even with 100 tetrahedra) is probably short compared to the grid updating phase which is sequential.

Medium grain. The following table presents the performance of the medium grain parallelization, i.e., two processes check the same object pair. The test was done with only two objects; no rendering, average on 5000 frames[5]. The arbitrary and box-tree algorithm were tested:

[4] Invocation:
`movem -x 5 -m +10 -s n 8 -a ar -t 10000 sh -l` *procs*
`movem -x 10 -m +10 -c -s g 8 -a cx -t 1000 sh -l` *procs*
`movem -x 10 -m +10 -a bx -s g 8 -t 1000 sh -l` *procs*
`movem -x 1 -m 100 -a ar -s g 8 -t 1000 sh -l` *procs*

[5] Invocation: `movem -x 20 -a ar -s n 8 -t 5000 sh -e 1.0 [-l 2]`

polygons	speed-up	
	arbitrary	box-trees
2×42	1.15	
2×132	1.73	
2×442	1.76	1.83

Concurrent algorithms. I tested the combination of the arbitrary algorithm tied together with the separating planes algorithm. The separating planes algorithm keeps and updates the planes for every object pair. Results have been very contradictory!

The first test involved only two objects, once with 2×100, once with 2×1000 polygons; it was carried out on an Onyx with 2 R4400 (150 MHz)[6]:

obj.types	speed-up	
	2x100	2x1000
Tori	1.5	3.5
Hys	2	1.9
Spheres	1.75	2.8

This is about what we would expect.

But when I tested many objects in the well-known cage, results were completely reverse! The set-up involved 10 objects, once with about 4000 polygons, once with 700 polygons altogether[7]. No rendering was done, architecture: Onyx with 2 R4400 (150 MHz).

complexity	frames/sec	
	with	without
4000	6.2	8.5
700	66	83

This is even stranger, since further tests showed that with this set-up about 80% of all collision queries were determined by the separating planes algorithm before the arbitrary algorithm, i.e., the concurrent algorithms should be definitely faster (on average) than just the arbitrary one alone.

More tests were done to find out what happened here; one of them tried to measure the time taken from the point where one of the algorithms found a solution until both algorithms returned. One side seems to be a problem: whenever the separating planes algorithm found a solution (i.e., "no collision"), a timer was started, and stopped when the convex algorithm returned (i.e., this is the time the convex algorithm did not "listen" to the other algorithm). This time seems to be way too long: 7.2 msec on average. But then, profiling should have shown that quite some time was spent in the barrier, but it didn't...

[6] Invocation: `movem -x 10 -t 30000 hy -a ar -s g 8 -e 1 -n` or `movem -x 33 -t 2000 sh -a ar -s g 8 -e 1 -n`, rsp.

[7] Invocation: `movem -x 10 -m +10 -a ar -s g 8 -n`, and `movem -x 5 -m +10 -a ar -s g 8 -n`; press the F1-key immediately after start.

Chapter 7

Implementation and Interface

Before we will present the overall outline of the collision detection module, we first need a few other pieces of the puzzle in order to provide a full functionality needed by real applications.

7.0.1 The Collision Interest Matrix

An application might be (and usually is) *only interested* in the collision of *some* objects [CAS92, GASF94]. For example, if a virtual reality application implements object manipulation by detecting collisions between a virtual hand and the objects of the scene, then this application will be interested only in collisions of the virtual hand (with other objects).

Furthermore, it is usually *only interested* in collisions between *certain pairs* of objects. For example, it is probably not interested in collisions between the fingers of a virtual hand; or, it is not interested in the collision between two adjacent links of a robot arm, because they do "collide" all the time.

Finally, an application has *additional information* about the objects of the scene; in particular, it is very likely to "know" that certain pairs of objects cannot collide at all (because the application sets their positions). For example, objects which are always fixed relative to each other cannot collide; however, only the application can have that knowledge. Letting the collision detection module know these facts can save time, especially if the objects are constantly very close to each other.

Altogether, we need some data structure to hold the relation which objects are "collidable". The easiest (and fastest) one is a matrix; it will be called the *collision interest matrix* or short *collision matrix*. One could use other data structures, like an individual list of "collidable" objects attached to each object. One could try to improve this method a little bit by providing different types of lists: a "collidable" or a "non-collidable" one. This could save quite some memory and time, if objects are "collidable" with all others.

However, we implemented the matrix method, because it is fastest and gives greatest flexibility. For example, it can hold certain data with every individual pair of objects: a callback function, the plane of the latest call to the separating planes algorithm (see Section 3.4.2).

Of course, we have to store only one half of the matrix, since it is symmetrical. We can do that, because we can sort the two indices (= object pointers); then we access an element of the matrix always with the smaller index for rows, and with the higher one for columns.

Memory doesn't seem to be a problem: in the current implementation, the matrix of 100 objects would need about 200 kBytes.

The collision matrix will be evaluated after the space indexing stage (see Section 7.1). This will filter out any pairs, which don't have to be checked for collision because the application is not interested, but which are close enough to each other, such that they are considered "potentially colliding".

7.0.2 Buffers Between Application and Collision Module

We have, of course, an array which contains all objects whose collision the application is interested in. This is the array holding all *collidable objects*. But not all of them will move every frame; it would be unnecessary work to process even those objects which haven't moved. So it is necessary to keep another array, which will hold all *moved objects* (we'll call it "moved-array" occasionally), i.e., whose transformation has changed since the last frame. Of course, the process of entering objects into this *moved* array is hidden from the application; it is hooked into the macros through which the application sets transformations.

We don't want an object to be inserted in the moved-array more than once, if the user happens to change the transformation several times between frames (which is usually the case). We also don't want to search the whole array before entering it. So we use another flag; this flag can be easily reset while taking objects out of the moved-array one by one and checking them for collisions.

Whenever a collision is detected, a callback is executed (this provides greater convenience for the application). However, the application might set new transformations in the callback, which would lead to an immediate re-insertion of the two objects into the moved-array. This could result in loss of the object or never-ending loops (depending on the implementation). In order to prevent these unpleasant effects, we need two moved-arrays, acting very similar to front- and back-buffer of the frame buffer: while the collision detection module removes one object at a time from the *front moved-array*, the application inserts new objects in the *back array* through setting their new transformations.

Concurrent mode

With serial execution of application and collision detection, the callback, processing pairs of objects which collide, could process these objects immediately, or it could just store them in a list, so that the application can process them later.

If the collision module runs concurrently to the application, and the callback processes objects immediately, then everything is just as with the serial case — the application doesn't have to bother whether the collision module is running concurrently or not.

If, on the other hand, the callback[1] stores objects just in a list, then we need to protect access to this list by another lock. Otherwise, their would have to be some synchronization mechanism (e.g., a barrier) which would tie the two loops (application and collision module) together, and which would inevitably slow down performance considerably. Of course, this buffer should be implemented by back- and front-buffer. The coupling is shown in Figure 7.1.

[1] remember that the callback is part of the application's code, but part of the collision's process!

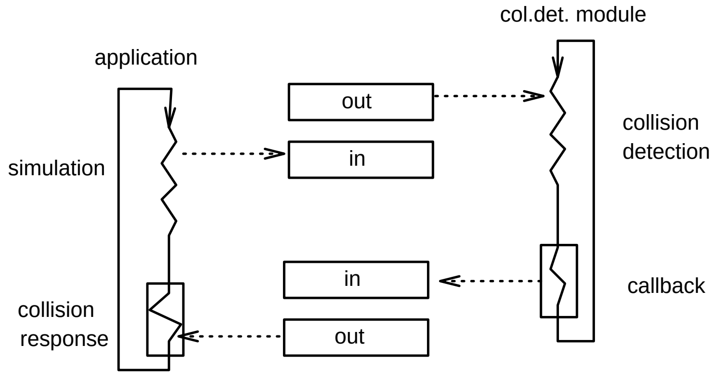


Figure 7.1. The coupling between concurrent application and collision detection module for the special case where the collision response callback stores colliding objects in a list to be processed later by the application.

7.1 Overview of the Module

All parts presented so far are integrated in the collision detection module. Most of them are hidden from the application, all of them are accessible only through functions (see Section 7.2).

The *front end* handles all requests, keeps track of "collidable" objects, and handles all underlying sub-modules (see Figure 7.2).

During the application loop, objects are being moved or their geometry changes. Whenever this happens, the object handler will hand this object to the collision detection module, which in turn adds it to an internal list of objects which have moved since the last frame. Of course, an application may move an object several times during the same frame — the object will be added to the list only once. The same will happen if the geometry ($\rightarrow$ the bounding box) of an object has changed. This is all hidden from the application by object handling macros/functions which have to be used in order to move or change an object.

At some point, the application will be *finished moving* all objects; it has to inform the collision detection module about this. At this point, the collision detection module will report all "new" collisions which have occurred since the last frame. By a new collision we understand a collision between objects which have moved or between a stationary object and a moving object.

The collision module front will first process all moved objects and *update the space indexing* data structure. It will use that space indexing structure which the application has chosen at initialization time (or none, if the application has chosen to do so).

After that, the front end will process the list of moved objects once more and, additionally, the list of all "collidable" objects, in order to find all *potential collisions.*

These will be filtered by the *collision interest matrix.* Only those pairs which the application is interested in will then be handed to one of the pairwise exact collision detection algorithms.

Currently, the *pairwise algorithm* employed is determined by the application at initialization time, i.e., it is the same for all pairs. With the *topological type* of objects, which is already available, and their *complexity*, it is easy (and will be implemented soon) to choose the optimal (i.e., fastest) algorithm for a given objects pair.

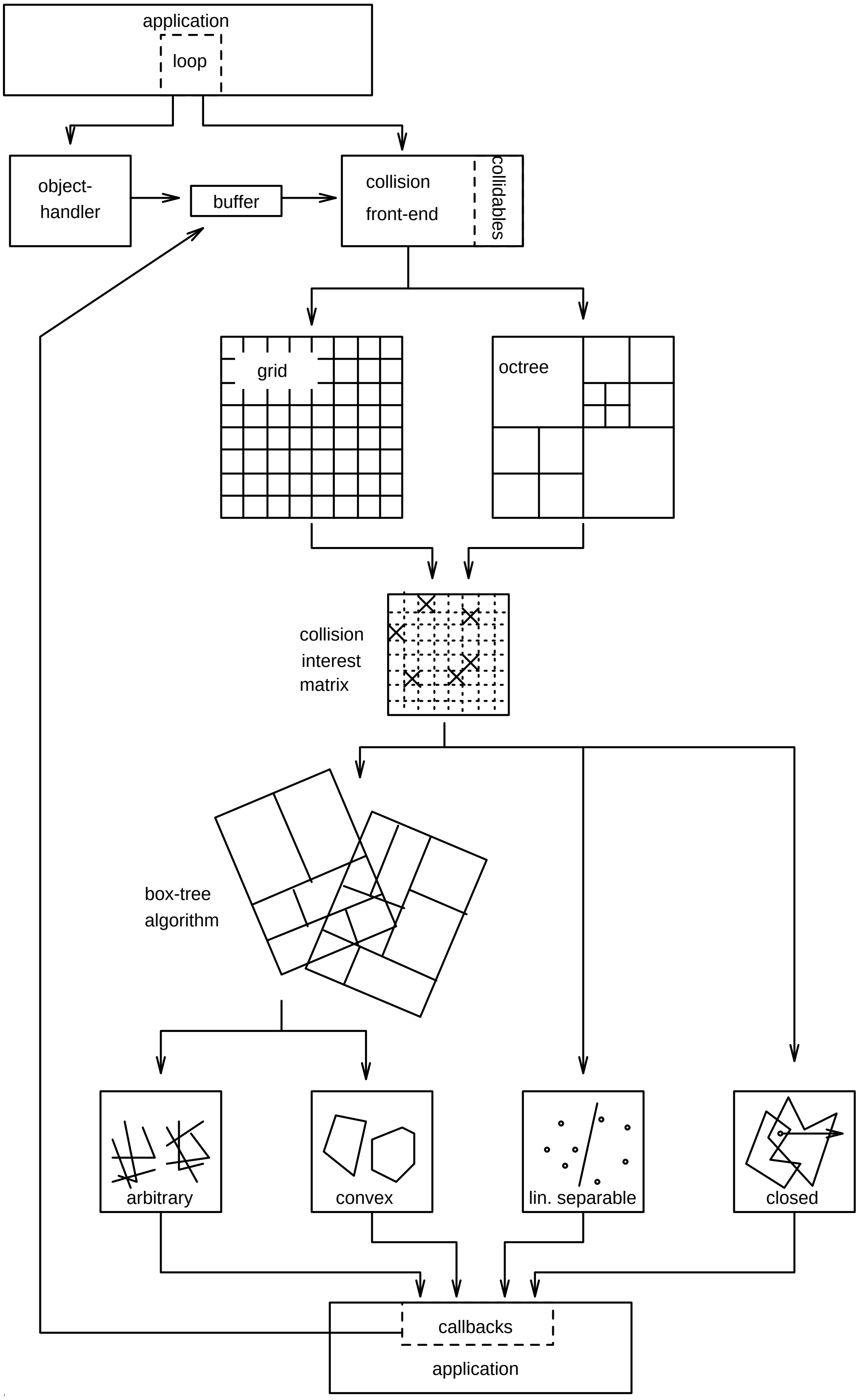


Figure 7.2. Integration of all parts of the collision detection module. Flow of control (serial execution).

If the pairwise exact algorithm finds a collision, it will execute an *individual callback* for each pair of objects.

The flow of data described above constitutes (conceptually) a *pipeline*, which is sketched by the following enumeration for the box-tree algorithm:

1. array of moved objects;
2. pairs of objects which occupy the same cell;
3. pairs of objects which the application is interested in;
4. pairs of overlapping sub-boxes (= subsets of the set of edges/faces, resp.;
5. pairs of intersecting edges and faces.

7.2 Functional Interface

The interface is entirely provided by the module's front end. If you're using the Y system, the interface consists of three functions (plus two more to set callbacks): an initialization function, a function to make objects "collidable", a start-up function, and a request function (during the loop).

An outline of the application will look like

```
Application outline
yuInitY( configuration )                                {calls colInit }
yuExitCallback( your-exit-fct )
create object hierarchy
colMakeCollidable( objects which application is interested in )
colStart()
loop:
    move objects
    colCheck()                    {will eventually call app. callbacks }
yuExitY()
```

In the case of concurrent collision, `colCheck()` doesn't do anything, since a never-ending collision detection loop is run by another process. The collision module will make sure, that an interrupt gets caught and will execute `yuExitY`, which in turn will execute *your-exit-fct*. Most applications will use `yuInitY(` *configuration* `)` and `yuExitY()` instead of the corresponding `col`...-counterparts.

See Section 9 for an outline of an application that also uses Into for input.

Description of functions

The following function is not provided by the collision detection module, but by the overall Y system. It is the initialization front-end; only parameters concerned with collision detection are discussed here. Below you'll find a full description of the `colInit`-function.

yuInitY

```
int yuInitY
( char *windowname, int argc, char *argv[], int winoptions,
vmmPointT universemin, vmmPointT universemax,
```

`colConfigE config, int maxnumcollidables, )`
Parameters:
`config` sets the configuration mode; can be or-ed of the following:

- `colOctreeIndex`, `colGridIndex`, `colNoSpaceIndex` select space subdivision: octree, regular grid, or none at all
- `colNonAlignedBBox` use non-axis-aligned bounding boxes for determining occupied cells of the space subdivision
- `colMethodArbitrary`, `colMethodConvex`, `colMethodNone` select exact collision detection method: for arbitrary polyhedra, for convex polyhedra only, or no exact collision detection at all
- `colMethodBoxtree` use an additional box-tree structure, can be combined with `colMethodArbitrary` or `colMethodConvex` (currently, however, only with `colMethodArbitrary`)
- `colParallel`, `colConcurrent` determines mode of parallel execution: parallel check of several pairs of objects, or concurrent collision detection loop
- `colParallSepPlane` run two algorithms in parallel for each object pair: one of the exact ones (specified above) and the separating plane algorithm.

`maxnumcollidables` the max. number of objects of which the application is interested in collisions, 0 = default (some large number). (With a much higher implementation effort I could have done without this parameter.)
`universemin`,`universemax` the extent of the universe.
Description: Initialize all data structures, in particular those needed for collision detection. The space indexing sub-module is initialized to work inside the box `universemin`, `universemax`.

This function sets the number of processes (if any parallel execution is wanted) and the resolution of the space indexing method (if any) to defaults. The default for the #processes is #processors.

The following functions are provided by the collision module itself. Any application needs them probably.

colMakeCollidable

`int colMakeCollidable( objPolyhedronP obj, void *clientcolldata )`
Parameters:
`obj` the polyhedron which the application is interested in any collisions of.
`clientcolldata` this pointer will be passed on to the callback in case of a collision.
Description: Make `obj` known to the collision detection module, i.e., make it "collidable". Only allowed after `colInit` but before `colStart`. Checks if the `obj` can be used for collision detection.

Does not create an edge list and doesn't classify objects! (This is done by `colStart`.)

colStart

`int colStart( void )`
Description: Start collision detection. Call this function after all objects have been made known to the module.

Classify topological type of objects. Insert them into the space indexing data structure.

If `colParallel` or `colConcurrent` are set, fork appropriate number of processes and have them idle wait. The interrupt signal handler is set to `yuExitY` (so, hook your own exit routine in by `yuExitCallback(`*own-exit-fct*`)`).

colSetCallback

`void colSetCallback( objPolyhedronP obj1, objPolyhedronP obj2, colCollisionCallbackT cb )`
Parameters:
`obj1`, `obj2`, pair of objects whose callback is to be set
`cb` their going to be callback
Description: Sets the individual callback of the object pair (`obj1`,`obj2`) to be `cb`. This callback will be executed whenever `obj1` and `obj2` collide with each other.

In case of a collision between objects P and Q, the callback `callbackPQ` associated to (P, Q) will be called by

```
callbackPQ( P, Q, clientdataP, clientdataQ )
```

or

```
callbackPQ( Q, P, clientdataQ, clientdataP )
```

colSetGeneralCallback

`void colSetGeneralCallback( colCollisionCallbackT cb )`
Parameters: `cb` the callback for all objects pairs.
Description: Sets the callback for all object pairs who do *not* have a callback *yet*.

If there will be many objects who will have the same callback, but a few who will have a different one, it's best to use `colSetGeneralCallback` first, then use `colSetCallback` for the few different ones.

colCheck

`void colCheck()`
Description: Detect all *new* collisions among "collidable" objects. A collision is "new" if at least one of the two objects has moved. (Actually, this is a macro, which doesn't do anything if the collision module runs in concurrent mode.)

The following functions are only needed if the application doesn't use the standard init- end exit-functions of the Y system (which is highly recommended).

colInit

```
int colInit
( colConfigE config, int resolution, int processes,
  int maxnumcollidables,
  float minx, float miny, float minz,
  float maxx, float maxy, float maxz )
```

Parameters:
`config` sets the configuration mode; can be or-ed of the following:

- `colOctreeIndex`, `colGridIndex`, `colNoSpaceIndex` select space subdivision: octree, regular grid, or none at all
- `colNonAlignedBBox` use non-axis-aligned bounding boxes for determining occupied cells of the space subdivision

- `colMethodArbitrary`, `colMethodConvex`, `colMethodNone` select exact collision detection method: for arbitrary polyhedra, for convex polyhedra only, or no exact collision detection at all
- `colMethodBoxtree` use an additional box-tree structure, can be combined with `colMethodArbitrary` or `colMethodConvex` (currently, however, only with `colMethodArbitrary`)
- `colParallel`, `colConcurrent` determines mode of parallel execution: parallel check of several pairs of objects, or concurrent collision detection loop
- `colParallSepPlane` run two algorithms in parallel for each object pair: one of the exact ones (specified above) and the separating plane algorithm.

`resolution` resolution of the space subdivision: 0 is a default, n corresponds to n^3 cells; for octrees, n will be rounded to the next lower power of two.
`processes` is the number of processes the module can use (with parallel execution), 0 = default = # processors.
`maxnumcollidables` the max. number of objects of which the application is interested in collisions, 0 = default (some large number). (With a much higher implementation effort i could have done without this parameter.)
`minx`,...,`maxz` the extent of the universe.
Description: Initialize all data structures needed for collision detection. The space indexing sub-module is initialized to work inside the box `minx`,...,maxz.

colExit

`void colExit( void )`
Description: Clean everything up, free everything. If the whole collision detection module has been compiled with `STATS_ON`, then also some statistics will be printed.

Exported variables

colCollisionEdge, colEdgePolyhedron, colCollisionFace, colFacePolyhedron

Whenever a collision is found, these variables will hold a *witness*, i.e., an edge of polyhedron `colEdgePolyhedron` and a polygon of polyhedron `colFacePolyhedron` which do intersect each other.

These variables are not (yet) reliably valid when parallel collision detection is on.

colArena

An arena out of which locks (and other stuff) can be allocated; this is exported just for convenience (an application using the concurrent collision detection mode might need a lock).

The module in parallel mode

A few things are different (still) if the module uses parallel collision detection; some should be kept in mind in order to avoid confusion later.

All callbacks are executed strictly *sequential*. This may or may not be desirable, depending on the application. In the future, this can be switched on/off by the application.

The variables `colCollisionEdge`, `colCollisionFace`, `colEdgePolyhedron`, `colFacePolyhedron` are *not valid*! (This is due to the fact that they're global) However, this will be mended in the future.

Concurrent mode

In this mode, everything is quite the same as in serial mode from the application point of view, *except* if the application's collision response callback just inserts colliding objects into a list, which is to be dealt with later by the application process (keep in mind that the callback is part of the collision process). In that case, the application has to establish a buffer as proposed in Section 7.0.2, page 116.

The swapping of the "in"- and "out"-buffers has to be done by the application. This needs to be protected by a lock (or similar). For convenience, the collision detection module provides an *arena* `colArena` (type `usptr_t`) out of which this lock can be allocated.

For an example, see the Y-Potter `yp.c`. In fact, this is also an example that it's not necessarily the objects which are buffered. Still, the buffers holding the items derived from colliding objects have to be swapped and access has to be exclusive.

A general outline of an application using this special arrangement of code would look like the following:

```
Application with concurrent collision module
AND separate collision response

in = buffer filled by callback
out = buffer where response function takes items out
L = access lock to buffers in/out

callback( a, b ):                                {add objects to list }
    acquire L
    add a, b to in
    release L

loop:
    move objects
    acquire L                                    {collision response }
    swap in and out                                     {out is empty }
    release L
    process objects in out
```

7.3 Implementation Details of Selected Algorithms

7.3.1 Time-Stamps

Very often, a speedup can be gained simply by keeping some *flags* with certain entities, which *mark their validity*, so as not to calculate things more often than necessary.

These flags have to be reset (to "invalid"), whenever the corresponding entity is rendered invalid. With reference to collision detection, in almost all cases this happens when an object has been moved, or (even worse) when it changes geometry.

However, it is way to expensive to loop over all those entities of an object which have become invalid. Instead, we introduce a counter for each class of flags (e.g., one counter for all face normals of an object). Then, when all of the entities of an object have become invalid, we simply increment that counter; when we make one of them valid again, we copy the value of that counter to the flag of the entity. A few examples are given below.

World face normals. For collision detection, face normals of objects have to be transformed into world coordinates[2]. This should be done only once after the object has moved, even though this object is handed very often to the collision detection routine.

The naive approach, without the time-stamp technique, spent 30% of the total CPU time with clearing all the flags for face normals in world coordinates. With the time stamp technique, this part of the collision detection module didn't show up any more in profilings.

Generating object pairs. When using some sort of space partitioning (like grid or octree), the same pairwise collision query could be generated several times, because both of the two objects share several cells. To prevent this from happening, we need to mark them somehow.

So we introduce a new time-stamp (let's call it *query* time-stamp) which will be incremented each time a collision query for any object O is done (so, it's basically a collision query counter). The value of this time-stamp will be attached to O. When looking for all object pairs (O, X) which might collide, we attach the same value to X whenever (O, X) is handed to some exact collision detection function. If X has this value already, we know that we checked (O, X) already during the current collision query with object O.

So far, the query time-stamp prevents only generating the same pair more than once during one collision query with a certain object O. Still the same pair (P, Q) could be generated twice during one whole collision detection phase: one time when checking for collisions with P, the other time when checking for collisions with Q.

So we need another time-stamp (let's call it *frame* time-stamp), which will be incremented with every frame. Whenever an object has been processed completely, we know there can't be any more collisions (for the current frame). So, we give it the value of the frame time-stamp. If we are about to check for collision of the pair (P, Q) and Q has got the value of this time-stamp already, we know that we don't have to look for a collision, because we have found all possible collision of Q with any other object.

7.3.2 Efficient Coding

All algorithms have been coded as efficient as possible. This includes

- loop unrolling for small ranges; the compiler can do loop unrolling only if the loop range is known at compile time.

  Examples are the test whether a point is in a polygon, the calculation of a polygon's bounding box, the calculation of a polygon's normal, etc.

- statistics gathering; this can degrade performance significantly if it is done in inner loops, even if it is only a simple counter incremement (in one case, this made up 17% of the total CPU time). Therefore, all statistics gathering code has been put inside `ifdef`'s, so that the optimized library doesn't contain any statistics code.

- floating point arithmetic; this is considerable faster than using doubles. However, care must be taken: the compiler must be told to not convert intermediate results to double format, and floating point constants must be given in a format, so the compiler does actually store them in single precision format.

[2] Of course, if there are only two objects which could possibly collide, then it's more efficient to transform the smaller set of face normals into the coordinate system of the other object

- interval computations (see Sections 3.2 and 3.6.1); the most often executed expression there would be

  ```
  if x < xmin then x = xmin;
  if x > xmax then x = xmax;
  ```

  However, if we exploit the fact that xmin < xmax is always valid, we can code the same like

  ```
  if ( x < xmin )
      x = xmin;
  else
      if ( x > xmax )
          x = xmax;
  ```

  which uses 1.3 comparisons less on average.

  When computing the minimum $\min(a, b)$ and maximum $\max(a, b)$ of two values simultaneously, we can save one comparison, by coding this like

  ```
  if ( a < b )
      min = a, max = b;
  else
      min = b, max = a;
  ```

It is usually not worth to extract common expression (arithmetic or pointer), because the compiler seems to be pretty good at that.

7.3.3 Collision Detection Among Arbitrary Polyhedra

The test whether an edge intersects a polygon (see Section 3.2) should be implemented as *lazy* as possible, i.e., computations are done *only when needed* and results should be *re-used* as much as possible. Thus, the part which tests whether an edge $e = (u, v)$ intersects a polygon f with normal n would look like:

$\boldsymbol{w} := \boldsymbol{v} - \boldsymbol{u}$
$a := \boldsymbol{w} \cdot \boldsymbol{n}$
$a \approx 0 \quad \longrightarrow \qquad \uparrow$ `‘‘no collision’’`
$\;b := \boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{u})$
$a < 0 \;\wedge\; (b > 0 \vee b < a) \quad \vee$
$a > 0 \;\wedge\; (b < 0 \vee b > a)$
$\;\longrightarrow \qquad \uparrow$ `‘‘no collision’’`
$\;t := \frac{a}{b}$
$\boldsymbol{x} := \boldsymbol{u} + t(\boldsymbol{v} - \boldsymbol{u})$
check if $\boldsymbol{x}$ inside the polygon f

By first calculating the nominator a and denominator b of $\frac{\boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{u})}{\boldsymbol{n} \cdot (\boldsymbol{v} - \boldsymbol{u})}$ and then testing for

$$
\begin{aligned}
&a < 0 \;\wedge\; (b > 0 \vee b < a) \quad \vee \\
&a > 0 \;\wedge\; (b < 0 \vee b > a)
\end{aligned}
$$

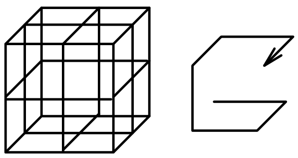


Figure 7.3. For the case where all sub-octants have to be visited, this traversal scheme minimizes assignments of octant bounds. Similar traversal schemes are used for the cases where less (at least $\leq \frac{1}{2}$ sub-octants have to be visited.

we can save a floating point division in the many cases where $t = \frac{a}{b} \notin [0,1]$. For calculating the point of intersection of edge e and polygon f, we can re-use the vector subtraction $\boldsymbol{v} - \boldsymbol{u}$.

Because this test is not called too often by the collision detection function of Section 3.2, it is not worthwhile to make it an in-line function (a macro in C).

Pre-check "edge bounding box intersects object bounding box".

7.3.4 Octree Algorithms

An efficient implementation of the basic insertion algorithm for octrees has to take advantage of the following simple conditions:

$$\begin{array}{lcl} \boldsymbol{b}_x^{\min} > \boldsymbol{o}_x^{\mathrm{mid}} & \rightarrow & \textit{don't consider the 4 left sub-octants at all} \\ \boldsymbol{b}_x^{\max} > \boldsymbol{o}_x^{\max} & \rightarrow & \textit{don't consider the 4 right sub-octants at all} \end{array}$$

where $\boldsymbol{b}^{\min}$, $\boldsymbol{b}^{\max}$ is the object's bounding box, and $\boldsymbol{o}^{\mathrm{mid}}$ is the center of the current octant. Similar tests with y and z yield a decision tree of depth 3. By explicitly implementing every case individually, we can take advantage of these conditions: for each leaf of the decision tree we can use a different sub-octant traversal scheme which minimizes assignments (see Figure 7.3). On some architectures, this gave a speed-up of 20%.

I tried to include an octant's bounds and its midpoint in the node structure, but that didn't help; not even when coordinates were still floats. So we can as well compute the bounds of sub-octants at recursion time, which saves about *half of the memory* required by the octree back-bone (the back-bone of a complete octree of depth 5 needs about 1-2 MBytes)!

The octree backbone, i.e., all octant nodes, are created at initialization time; they remain throughout the whole run time. This is done to minimize `malloc`s and `free`s. Object arrays are also created at init.-time, but only very small ones; their size can *grow and shrink* (which is done by `realloc`), depending on how many objects are in an octant. However, this growing/shrinking is controlled by some hysteresis, so the number of calls to `realloc` is kept low. Tests showed that by increasing this Hysteresis from 5 to 10, the number of `realloc`s was decreased by a factor 3; however, the overall running time did not decrease significantly, because `realloc`s seem to be fast compared to an octree traversal.

7.4 Lessons learnt

I learnt doing timing tests and optimization the hard way... Some of the lessons I learnt I would like to summarize here; of course, in retrospective most of them are quite obvious...

Timing.

- Make tests *immediately before* you modify the implementation; afterwards, do the *same tests* again.

  You have to know in advance how you're going to evaluate the improvements, so you can do the tests with which you have to compare the tests afterwards (if you don't exactly know, what/how to time, keep the old source code). Keep at least the old executable.

- Do the whole experiments on different architectures! (see "Optimization")

- 1000 frames are *not* enough! (Even if they take hours...) Especially when randomness is involved (like random starting positions). If possible, remove any randomness when comparing timings.

- Make sure that there are exactly the *same conditions* with all tests! Of course, it's best to compare results obtained on the same machine at he same time.

  This includes:

  - same compiler options, like `-O`, `-float`, etc.
  - same CPU, same architecture, same clock rate, same OS,
  - check system performance meter (like `gr_osview`, `top`) during the test; check CPU, memory usage, swap waits, ethernet activity, interrupts, and graphics; make sure the performance meters don't gobble system resources: `ps` locks memory for quite some time, `gr_osview` and `xclock` can use huge amount of graphics and CPU resources on some machines!
  - make sure the load is the same,
  - make sure the machine is not an NIS server, or something similar

- Don't do any graphics; this causes lots of waits.

- Remove any *statistics gathering code*; a simple counter in an inner loop (like one of mine in the edge-polygon-intersection test) could easily cost 10% of the time.

Optimization.

- Always make timing tests to evaluate your "improvements"! Check the optimization on different architectures! There are quite some surprises in stock...

  For example: the 2×2 pointer scheme with grids would have accelerated the algorithm a lot, *if* they hadn't caused a tremendous amount of *cache misses*! Another example: the box-tree algorithm ran twice as slow as the arbitrary collision algorithm on the VGX, but twice as fast on Onyx and Indigo!

- The `-O` compiler option helps only on some architectures! On others it might even slow things down.

- *Profile* the code, before optimizing! Most of the time, I was astonished.

  Unfortunately, profiling helps only identifying hot spots, but not finding bad algorithms. Also, profiling results are very different on different architectures.

- Some hints:

- don't try to be too clever: `a[i]=...; i++;` offers the compiler more opportunities than `a[i++]=...;`.
- don't `memcpy()` for less than 16 bytes; `a[0]=b[0]; a[1]=b[1]; a[2]=b[2];` is better than `memcpy(a,b,12)`.
- don't use large (100,000s) local static arrays, make them global static instead; (I'm not sure of this, but my notes say that some piece of code was slowed down by a factor of 20 with declaring an array local static instead of global static...)
- for ultimate optimization: look at the assembly code first. Sometimes the compiler's better than you are, sometimes not.

Parallelization.

- Do not use any global variables.
- The only debugger for parallel code is `printf`.
- If you think that you don't have to guard an exclusive region because it is very unlikely to happen — be sure it will happen!

  (For example: swapping the two moving-object buffers between producer (application) and consumer (concurrent collision detection module) is done with 3 pointer assignments; when there were only a few objects, nothing happened, but with 100 moving objects, all of a sudden some of them escaped the cage!)

Chapter 8

The Potter – an Application

One of the first applications using the exact collision detection module is a rewriting of an existing experimental application, the *Potter*. The basic idea of this application is to be able to modify the geometry of an object using a virtual input device like data glove, space mouse, or other. Up to now, the main focus has been on determining collisions fast enough. In the future, I hope to be able to improve modeling algorithms.

The tool is a virtual hand, right now; it could be any other tool either. The Potter can be configured at start-up time (by command line option) to use either a single finger tip, all finger tips, the index finger, or the whole hand for modeling. Figure 8.1 shows a torus after being modeled.

Currently, polygons are moved always "inside", which is determined by their normal. Of course, better modeling modes would be very desirable — like dragging,

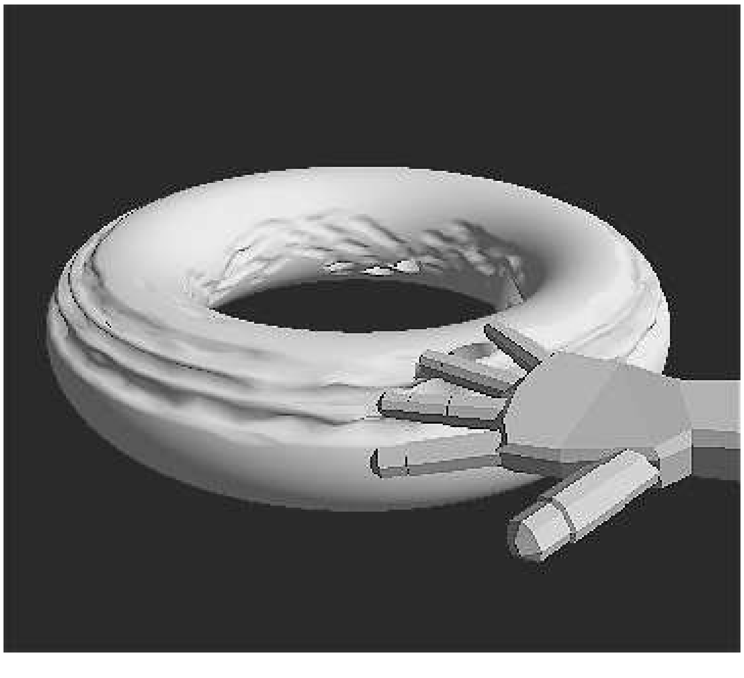


Figure 8.1. A torus having been modeled by a virtual hand.

applying weigh-functions to the surrounding polygons [Bryson92], or even splitting objects into two.

The Potter (called "Y-Potter" in contrast to the old one) takes full advantage of all collision detection module options. It can be run in plain serial mode, in concurrent mode, or using parallel collision detection of multiple collisions. Since geometry changes, and since the "pot" is not convex most of the time, it has to use the arbitrary algorithm (see Section 3.2). But it does offer to use additional algorithms: the concurrent separating planes algorithm (see Section 3.4.2) and the relaxed polygon collecting phase (see Section 3.2.1).

The initial "pot" can be chosen from a variety of shapes: sphere, torus, cylinder, tetra-flake, hyperboloid. The resolution of them can be chosen to be anything from 10 polygons up to 10,000 polygons. The geometry is originally mostly quadrangles (except for the tetra-flake) — but it can be triangulated.

8.1 A Simple Modeling Algorithm

The approach taken here to modify the surface of the pot is to move polygons which are hit by a part of the hand "inside", which is opposite the direction of their normal.

The collision detection module reports edges and polygons if there is a collision. However, it could be either the polygon *or* the edge which belongs to the pot. So, the potter first collects all polygons and all edges which are reported by the module. Furthermore, the same edge/polygon could be reported several times. This could happen if there are several "active" parts of the hand, or if the collision detection runs concurrently and the detection loop is faster than the response loop (see Figure 7.1). So, before we add an edge/polygon to the buffer (the "in"-buffer in Figure 7.1), we check that it is not yet in there.

After all colliding edges/polygons have been collected, the edges are "translated" into polygons, i.e., the two incident polygons of a colliding edge are added to the "hit polygon" array.

Then, all polygons colliding with some part of the hand are processed. Each of them is moved a fixed fraction of the (normalized) normal in the opposite direction of it. In consequence, normals of adjacent polygons have to be re-computed, too.

After a polygon has been moved, and all normals of adjacent polygons have been made valid again, we have to re-compute vertex normals, since the pot is rendered with Gouraud-shading. All those vertex normals have become invalid which are incident to a polygon whose normal has changed.

Figure 8.2 shows which entities have to re-computed for a single colliding polygon. Currently, we do not take care of the case when polygons are indirectly modified several times. This could happen if there are several "active" parts of the hand, which hit polygons which are just one polygon apart from each other. However, this shouldn't be a problem (see below, "Avoiding cancer").

One reason for this new Potter version being much more efficient is (I believe) that the DCEL data structure enables us to find very quickly adjacent polygons and vertices (see Section 4.1.2). With a very simple loop we can enumerate all edges or vertices incident to a polygon, with another simple loop, we can find all polygons incident to a vertex. Thus we can find by two or three simple loops all polygons adjacent to the one being moved and all vertices whose normal has to be re-computed, resp.

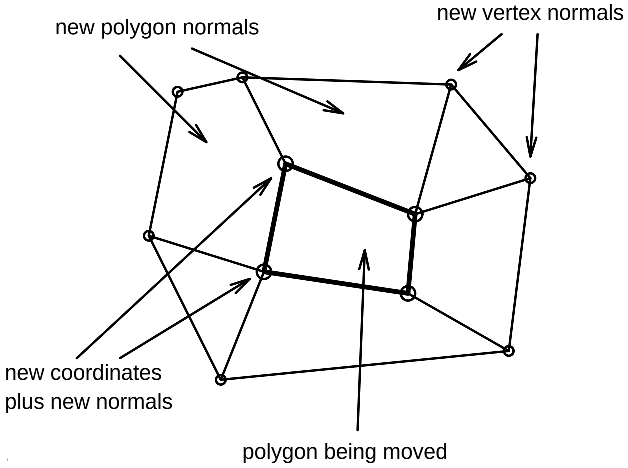


Figure 8.2. Geometrical features which have to be (re-)computed when a polygon is being moved

There are several ways to get at the vertices which are adjacent to the vertices of the polygon being moved. If we choose the right one, we can avoid computing vertex normals more than once: we simply have to loop over all vertices of the moved polygon, then loop over all edges incident to the current vertex, and pick the "other" vertex of the edges.

Special case: still pot. If the pot doesn't move, we can restore *all* information which is relevant to the collision detection algorithm. With a moving pot, the algorithm has to compute all vertices, normals, and bounding boxes in world coordinates. If the pot does not move, we can restore the world coordinates of those items during the modification ourselves.

This is much more efficient, since we know exactly, which normals, bounding boxes, etc. have become invalid. Thus, we save the collision detection module a lot of work.

If the pot does move, though, we need not bother restoring anything, because the collision detection has to compute everything anyway.

Avoiding cancer. A problem with the current approach to modify geometry is that the pot "breaks" pretty easily: after a while, some polygons might actually be pushed (apparently) "outwards". They are not really pushed outwards; instead, during the modeling process they got turned so much that their normals are actually facing to what the users thinks to be "inside".

I tried to avoid this by preventing any dihedral angle to get out of a certain range around $180°$, say $[160, 200]$. So after a polygon has been moved, all dihedral angles which have changed are stored in an array `I`. All edges in `I` are then checked if they're still in range — if they're not, their vertices are moved a little to the "inside" or the "outside" depending on whether the edge is "convex" or "concave", resp. Then all those face normals are re-computed which have changed by this correction. Also, the dihedral angle of some edges have changed: these edges are added to the array `I` again (if they're not yet there).

This array of edges, `I`, is scanned a certain maximum number of times for "bad" edges.

The method described above did improve the "cancer" effect a little bit. However, it could not completely alleviate the effect. It is not really clear to me why it does not. However, it might not be worth the effort, because entirely different methods how to modify geometry might be much more intuitive and user-friendly, apart from being more robust.

Other modeling methods. Further development of this application should focus more on the modeling part. Modeling methods should be intuitively clear and natural, apart from being robust. Some methods which seem to be promising are:

- Take the path of the point of collision on the hand into account, i.e., apply the displacement of this point to the polygons being hit. Maybe, the motion of the point of collision on the pot should be taken into account, too.
- Apply translational offset also to polygons in the vicinity of the colliding polygon weighed by some function.

  Take care that polygons do not "crumpled up", which might happen if they are pushed perpendicular to their normal and some vertices are moved further than others.

  Use tools with adjustable size, so as to provide for coarse and fine modeling. Make the "vicinity" the larger the bigger the modeling tool is.
- Preserve the volume of the pot.

8.2 Results

A few timings are presented here.

With concurrent collision detection the "frame rate" which the user perceives is only the time taken by application and renderer. Currently, this is 13-15 frames/sec with a pot of 10,000 polygons (not meshed).

The plain algorithm shows the following performance (incl. rendering and application):

#polygons	frames/sec	
	full hand (16 obj.s)	forefinger (3 obj.s)
9944	3-4	8-10
4220	10-13	20

The relaxed polygon collecting phase together with the improved method for the still pot case gives quite some speed-up:

#polygons	frames/sec	
	full hand (16 obj.s)	forefinger (3 obj.s)
9944	5-7	10-12
4220	15-20	23-28

Chapter 9

Collision Detection for Virtual Buttons

The collision detection module has been integrated with an already existing toolkit (called *Into*) providing logical input devices [FSZ94], which is used in an in-house virtual reality system [AFM93, Felger92].

Logical input devices.

Logical input devices are *abstractions* of physical input devices. Analogously to GKS, logical input devices provide device-independent input. There are several classes of logical devices:

- buttons produce binary input,
- choices have a discrete 1-out-of-n value,
- values produce a continuous scalar,
- both locations and orientations are (conceptually) 3-dimensional, with different representations,
- spaces consist of one location and one orientation, thus they are 6-dimensional,
- a hand is a 20-dimensional device, meant to be used for controlling virtual hands.

All of them can be used in event or in poll mode.

Logical devices are mapped on physical input devices by the application; usually one logical class can be mapped onto several different physical classes (e.g., a value can be mapped onto a physical mouse, a tracking sensor, a joint of a data glove, etc.).

Physical devices are handled by servers which Into communicates with via sockets; thus physical devices can be connected to any machine.

An overview of Into is shown in Figure 9.1.

Virtual input devices

A significant enhancement compared to conventional device abstraction is the introduction of virtual input devices. In general, virtual devices are logical devices which are mapped on other logical input devices. (Of course, eventually the chain will end at some physical devices.)

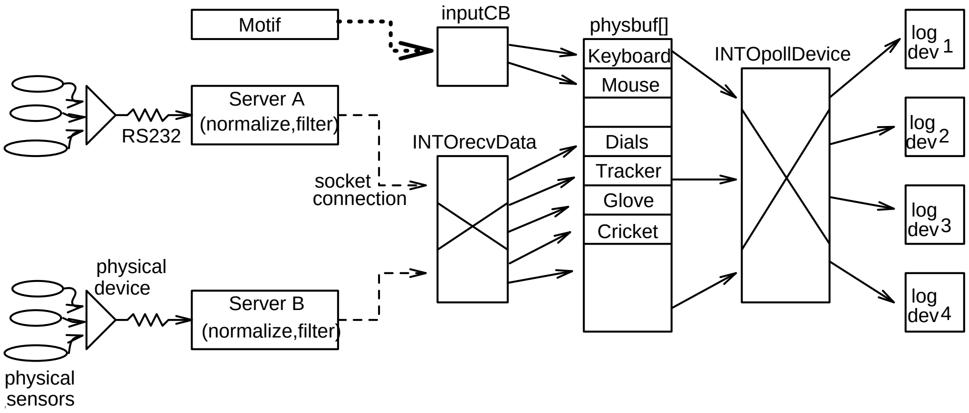


Figure 9.1. Overview of the input/output toolkit Into.

Of special interest in this context are virtual buttons. These are graphical objects, the *button object*, which can be "pressed" by another object, the *finger object*. The finger object in turn is controlled by a logical input device. A virtual button is considered "pressed" when the finger object and the button object collide.

Implementation

So, whenever Into has to determine the status of a virtual button, a collision check is done with the button and the finger object (conceptually). Consequently, there has to be a link between Into and the collision detection module. A monolithic integration does not seem to be worthwhile. First, the two modules are fairly independent by their mere concept and tasks; second, the Into module must be compilable for different renderers (currently, two are supported).

Essentially, there are two ways to link the two modules together. Either, the collision detection module provides a way to inquire about a collision between two particular objects; or, the Into module gets called back the collision module.

I chose to implement the second scheme, because this didn't require the interface to be changed. Furthermore, it doesn't matter where the collision status of button and finger object is stored. Certainly, we don't want to store the status of every object pair, because most applications won't be interested at all in that kind of collision inquiry.

Into's internal data structures have been extended, in order to hold the collision status of a finger-button pair. This status is stored with internal data for the virtual button (currently, a virtual button can be pressed by only one finger object).

So, collisions of a virtual button and its finger object are detected when the application queries the collision detection module for all collisions which have occurred since the last time. At this time, the collision module will execute a callback of the Into module which stores the event with the virtual button's internal data. Whenever the application polls the status of a virtual button, this data is just reported to the application.

Of course, the Into module will announce the finger and button objects to the collision module, so the application doesn't have to take care about that.

Application outline including Into

The outline of an application which also uses Into for input will have the following outline (for more details see Section 7.2):

```
Application with Into

yuInitY
yuExitCallback( ...)
create object hierarchy
INTOinit(...)
colMakeCollidable( objects which application is interested in )
                {virtual button objects will be taken care of by Into }
create logical input devices
colStart()
loop:
    move objects
    colCheck()          {will call application and Into callbacks }
    INTOevents()
yuExitY()
```

Chapter 10

Conclusion and Future Work

10.1 Conclusion

Several algorithms have been implemented for fast, exact collision detection on the level of object pairs. By utilizing many different pre-checks, conventional algorithms have gained considerable speed. Further speed-up has been gained by pre-phases which collect relevant polygons, and by lazy evaluation which avoids calculating normals or bounding boxes which are not needed.

Probabilistic algorithms have been developed to be tied together with conventional algorithms, thus yielding speed-up in cases where conventional algorithms are rather slow.

A new algorithm has been developed to solve the $O(n^2)$-problem on the edge-polygon level. It uses a hierarchical, adaptive data structure (the "box-tree") to discard quickly "un-interesting" polygons.

Few approaches so far have attempted to provide an integrated solution which addresses fast exact pairwise collision detection as well as fast retrieval of potentially colliding objects. In order to overcome the n^2-problem on the global level, algorithms for octree and grid have been developed; these have been designed with special regard to highly dynamic scenes. Several variants of both of them have been evaluated.

To my knowledge, no one so far addressed the issue of parallelizing collision detection. Three different parallelization schemes have been developed and implemented. A fourth one (fine grain) is described.

A simple, easy-to-use interface has been implemented which incorporates all algorithms developed with this thesis. Several levels can be configured at run-time: the pairwise algorithm to use, the space partitioning data structure (if at all), the collision interest matrix, and the mode of parallelization (if at all).

10.2 Future work

We will give here just a summary of all the directions and further improvements that could be taken in order to improve collision detection in highly dynamic environments. They have been presented in more detail in previous sections in the context to which they belong.

Pairwise algorithms. The algorithm for the most general polyhedra and the Cyrus-Beck algorithm for convex polyhedra seem to have reached their limits.

However, by applying *caching techniques*, storing results of previous detection queries, they seem to offer quite promising opportunities for further speed-up in environments where objects move slowly compared to the frame-rate (which is the main aim of every interactive visualization system).

Convexity should offer greater possibilities for speed-up than algorithms exposed so far. It might be that more efficient pre-checks can to be developed. In any case, I think that convexity offers big advantages for incremental techniques (see Sections 3.4.1, and 3.4.2).

The *probabilistic algorithms* of Sections 3.4.2 and 3.4 (using *separating planes* and the *point-in-polygon test*) seem to offer further improvements. Certain computations could be eliminated or arranged more efficiently with the separating planes algorithm. The box-tree data structure (see Section 3.6) could be used to speed up the point-in-polygon test (used by the second prob. algorithm). Both of the two algorithms should be tied together for a concurrent early decision whether two objects collide or not.

The *box-tree* algorithm seems to offer many more opportunities: among others are an "on-the-fly" axis-aligned traversal, empty sub-boxes within the tree, combined box-trees for edges and faces, further specialization of the cut-plane, and an adaption to convex objects.

It seems to me that *parallelization* could still yield greater speed-up (although it is already linear). With only very few processors available, there aren't too many further opportunities, except for reducing synchronization overhead and parallel updating of the space partitioning data structure. With more processors (some ten), there seem to be further possibilities: the parallel schemes developed so far can be applied at the same time; fine grain parallelization might be worthwhile, too.

Bounding volumes should be investigated further. There are several options which might lead to significant speed-up. Maximal boxes, which contain an object at any orientation; the advantage is that these boxes are very inexpensive to transform, and they need to be calculated only at start-up time. Also, tight bounding boxes might reduce the number of pairwise collision detection queries; they can be updated from previous frames efficiently if the object "inside" is convex.

Chapter 11

Acknowledgements

There are so many people which I have to thank so much, it is just impossible to do so. In fact, a complete list would contain almost all people I have ever really met.

So, instead of writing the longest section of this thesis, I will thank just those who have, in some way or other, participated in making this thesis.

I am grateful to Prof. Encarnação and Dr. Martin Göbel for making possible and organizing the great experience of staying half a year in the US.

I would like to thank my advisor Wolfgang Felger, with whom to work was my pleasure for four years, who put in very much time and effort to get me to the US, and who finally made possible my stay at the National Center for Supercomputing Applications in Urbana, Thanks also to Stefan Müller, who had quite some trouble, too.

I would like to thank also the many more people in Urbana, who made it possible that I did (most of) my thesis there: Donna Cox who was the first to give her consent; Melanie Loots, Shirley Shore, Jeanne Soliday, and quite a few others, who had quite a bit of administrative and organizational work.

Bill Sherman gets lots of thanks for willing to be my local advisor, giving immediate technical support whenever I needed it, and for generously lending me a TV.

Bettina and George Francis have been very kind by providing a regular event (the "Tuesday dinner") where I could meet people. In fact, in the beginning, this Tuesday dinner was the only event of the whole week where I actually did meet people. Also, it was the only meal of the week which was really substantial and it tasted just delicious!

I would also like to thank all folks at Beckman Institute for quite a few nice, casual talks (in random order): George Baxter, Ulrike, Kelly, Rachel, Robin, Milana. Thanks to you, Doug, for lots of distraction you caused by improving my "movie" education and playing Battlezone with me (:-)).

While I stayed at CRCG in Providence, all people in the office created a friendly atmosphere which was nice to work in (again in random order): Elaine and Peter Bono, Rajeev, Satish, Robin, Winfried, Jens, Brian, Eddie & Jutta, and Stefan.

My parents gave me very helpful day-to-day support during my stay abroad, which is not to be under-valued!

Many thanks go to Birgit, my soon-to-be wife, for her patience, and for proof-reading this text in many many hours (which must have been sort of a pain). Of course, I am responsible for any errors still remaining.

V

Bibliography

[ADG+94] Peter Astheimer, Fan Dai, Martin Göbel, Rolf Kruse, and Stefan Müller, and Gabriel Zachmann. *Realism in Virtual Reality*, pages 189–210. Wiley & Sons, 1994.

[AFM93] Peter Astheimer, and Wolfgang Felger, and Stefan Müller. Virtual Design: A Generic VR System for Industrial Applications. *Computers & Graphics*, 17(6):671–677, 1993.

[Arvo91] James R. Arvo, editor. *Graphics Gems II.* Academic Press, San Diego, 1991.

[AS90] P. K. Agarwal and M. Sharir. Red-blue intersection detection algorithms, with applications to motion planning and collision detection. *SIAM J. Comput.*, 19:297–321, 1990.

[BB88] Gilles Brassard and Paul Bratley. *Algorithmics; Theory and Practice.* Prentice Hall, 1988.

[BD92a] C. Bajaj and T. K. Dey. Convex decomposition of polyhedra and robustness. *SIAM J. Comput.*, 21:339–364, 1992.

[BD92b] C. Bajaj and T. K. Dey. Convex decomposition of polyhedra and robustness. *SIAM J. Comput.*, 21:339–364, 1992.

[BDG90] Jos/'e Luis Balcazar, and Josep Diaz, and Joaquin Gabarr/'o. *Strucutral Complexity I. ATCS Monographs on Theor. Computer Science.* Springer, 1990.

[BF79] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[BJ91] W. Bouma and G. Vanecek Jr. Collision Detection and Analysis in a Physical Based Simulation. In *Eurographics Workshop on Animation and Simulation*, pages 191–203, 1991.

[Bryson92] Steve Bryson. Paradigms for the Shaping of Surfaces in a Virtual Environment. In *Siggraph '92, 19th International Conference On Computer Graphics and Interaction Techniques, Course Notes 9*, pages 13.1–13.10, 1992.

[Canny86] John Canny. Collision Detection for Moving Polyhedra. *IEEE Transactions an Pattern Analysis and Machine Intelligence*, PAMI-8(2):200–209, March 1986.

[CAS92] Gregory M. Herb Clifford A. Shaffer. A Real-Time Robot Arm collision Avoidance System. *IEEE Transactions on Robotics and Automation*, 8(2), April 1992.

[CCV85] I. Carlbom, and I. Chakravarty, and D. Vanderschel. A Hierarchical Data Structure for Representing the Spatial Decomposition of 3-D Objects. *IEEE Computer Graphics and Applications*, 5(4):24–31, April 1985.

[CD87] B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *J. ACM*, 34:1–27, 1987.

[Chazelle84a] B. Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM J. Comput.*, 13:488–507, 1984.

[Chazelle84b] B. Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM J. Comput.*, 13:488–507, 1984.

[CM87] Yong C. Chen and Catherine M. Murphy. H-P Model — A hierarchical space decomposition in a polar coordinate system. In Tsiyasu L. Kunii, editor, *Computer Graphics 1987 (Proceedings of CG International '87)*, pages 443–459. Springer-Verlag, 1987.

[CP90a] B. Chazelle and L. Palios. Triangulating a non-convex polytope. *Discrete Comput. Geom.*, 5:505–526, 1990.

[CP90b] B. Chazelle and L. Palios. Triangulating a non-convex polytope. *Discrete Comput. Geom.*, 5:505–526, 1990.

[CT87] Min Chen and Peter Townsend. Efficient and consistent algorithms for determining the containment of points in polygons and polyhedra. In G. Marechal, editor, *Eurographics '87*, pages 423–437. North-Holland, August 1987.

[Dai94] Fan Dai. Private communication, 1994.

[Devillers88] Olivier Devillers. The Macro-Regions: an Efficient Space Subdivision Structure for Ray Tracing. Technical Report 88–13, Laboratoire d'Informatique de l'Ecole Normale Superieure, Paris, France, November 1988.

[DK83] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoret. Comput. Sci.*, 27:241–253, 1983.

[DK85] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6:381–392, 1985.

[Dyer82] C. R. Dyer. The Space Efficiency of Quadtrees. *Comput. Graphics and Image Process. (USA)*, 19:335–348, August 1982.

[Edelsbrunner94] Herbert Edelsbrunner. Private communication, 1994.

[EH72] D. J. Elzinga and D. W. Hearn. The Minimum Covering Sphere Problem. *Management Science*, 19(1):96–104, September 1972.

[ELP87] M. Erdmann and T. Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2:477–521, 1987.

[ES88] José L. Encarnação and Wolfgang Straßer. *Computer Graphics*. Oldenbourg Verlag, 3 edition, 1988.

[FA85] W. R. Franklin and V. Akman. Building an Octree from a Set of Parallelepipeds. In M. Wein and E. M. Kidd, editors, *Graphics Interface '85 Proceedings*, pages 353–359. Canadian Inf. Process. Soc., 1985.

[Felger92] Wolfgang Felger. How interactive visualization can benefit from multidimensional input devices. In J. R. Alexander, editor, *Visual Data Interpretation Proc., SPIE 1668*, 1992.

[FK85] Kikuo Fujimura and Tosiyasu L. Kunii. A Hierarchical space indexing method. In Tsiyasu L. Kunii, editor, *Computer Graphics Visual Technology and Art (Proceedings of Computer Graphics Tokyo '85)*, pages 21–33. Springer-Verlag, 1985.

[FKN80] H. Fuchs, and Z. M. Kedem, and B. F. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 124–133, July 1980.

[FSZ94] Wolfgang Felger, and Reiner Schäfer, and Gabriel Zachmann. Interaktions-Toolkit. Technical Report FIGD-94i002, Fraunhofer Institute for Computer Graphics, Darmstadt, January 1994.

[FvDFH90] J. D. Foley, A. van Dam, and Steven K. Feiner, and John F. Hughes. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Publishing Company, second edition, 1990.

[GA93] I. Gargantini and H. H. Atkinson. Ray Tracing an Octree: Numerical Evaluation of the first Intersection. *Computer Graphics forum*, 12(4):199–210, 1993.

[Gascuel93] Marie-Paule Gascuel. An Implicit Formulation for Precise Contact Modeling Between Flexible Solids. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 313–320, August 1993.

[GASF94] Alejandro Garcia-Alonso, and Nicolás Serrano, and Juan Flaquer. Solving the Collision Detection Problem. *IEEE Computer Graphics and Applications*, ?(?):36–43, May 1994.

[GJK88] Elmer G. Gilbert, and Daniel W. Johnson, and S. Sathiya Keerthi. A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, 1988.

[Glassner89] A. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.

[Glassner90a] Andrew S. Glassner, editor. *Graphics Gems*. Academic Press, San Diego, CA, 1990.

[Glassner90b] Andrew S. Glassner, editor. *Graphics Gems*, chapter An efficient bounding sphere, page 301 ff. In Glassner [Glassner90a], 1990.

[Glassner90c] Andrew S. Glassner, editor. *Graphics Gems.* Academic Press, 1990.

[Hahn88] James K. Hahn. Realistic Animation of Rigid Bodies. *Computers & Graphics*, 22(4):299–308, August 1988.

[Hay92] Ray
Tracing News. URL: ftp://princeton.edu:/pub/Graphics/RTNews, September 1992. electronic material.

[HKP91] John Hertz, and Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computing.* Addison-Wesley, 1991.

[HL92] Josef Hoschek and Dieter Lasser. *Grundlagen der geometrischen Datenverarbeitung.* B.G. Teubner, Stuttgart, 2 edition, 1992.

[HSS83] J. E. Hopcroft, and J. T. Schwartz, and M. Sharir. Efficient detection of intersections among spheres. *Internat. J. Robot. Res.*, 2(4):77–80, 1983.

[Hubbard93] Philip M. Hubbard. Interactive Collision Detection. In *IEEE Symposium on Research Frontiers in VR, San José, California*, pages 24–31, October 25–26 1993.

[Kalay82] Y. E. Kalay. Determining the Spatial Containment of a Point in General Polyhedra. *Comput. Graphics and Image Process. (USA)*, 19:303–334, August 1982.

[Kirk92a] David Kirk, editor. *Graphics Gems III.* Academic Press, Inc., San Diego, CA, 1992.

[Kirk92b] David Kirk, editor. *Graphics Gems III*, chapter A linear time simple bounding volume algorithm, page 301 ff. Volume 2 of Kirk [Kirk92a], 1992.

[KK86] Timothy L. Kay and James T. Kajiya. Ray Tracing Complex Scenes. In David C. Evans and Rusell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 269–278, August 1986.

[LB84] Y.-D. Liang and B. A. Barsky. A New Concept and Method for Line Clipping. *ACM Trans. Graphics (USA)*, 3:1–22, January 1984.

[LC91] Ming C. Lin and John F. Canny. A Fast Algorithm for Incremental Distance Calculation. In *Proc. of the 1991 IEEE International Conference onRobotics and Automation*, pages 1008–1014, April 1991.

[LC92] Ming C. Lin and John F. Canny. Efficient Collision Detection for Animation, September 1992.

[Lengauer90] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley, Teubner, 1990.

[Linhart90] Johann Linhart. A quick point-in-polyhedron test. *Computers and Graphics*, 14(3/4):445–447, 1990.

[LM91] Ming C. Lin and Dinesh Manocha. *Efficient Contact Determination Between Geometric Models.* PhD dissertation, University of California, University of North Carolina Chapel Hill, URL: ftp://ftp.cs.unc.edu/pub/techreports/94-024.ps.Z, 1991(?).

[LR82] Y. T. Lee and A. A. G. Requicha. Algorithms for Computing the Volume and Other Integral Properties of Solids. II. a Family of Algorithms Based on Representation Conversion and Cellular Approximation. *Commun. ACM (USA)*, 25:642–650, September 1982.

[Megiddo82] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 329–338, 1982.

[MMZ94] Jai Menon, and Richard J. Marisa, and Jovan Zagajac. More Powerful Solid Modeling through Ray Representations. *IEEE Computer Graphics and Applications*, pages 22–35, May 1994.

[MP78a] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[MP78b] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[MS85a] K. Mehlhorn and K. Simon. Intersecting two polyhedra one of which is convex. In L. Budach, editor, *Proc. Found. Comput. Theory*, volume 199 of *Lecture Notes in Computer Science*, pages 534–542. Springer-Verlag, 1985.

[MS85b] K. Mehlhorn and K. Simon. Intersecting two polyhedra one of which is convex. In L. Budach, editor, *Proc. Found. Comput. Theory*, volume 199 of *Lecture Notes in Computer Science*, pages 534–542. Springer-Verlag, 1985.

[MSH+92] M. D. J. McNeill, B. C. Shah, M.-P. Hébert, and P. F. Lister, and R. L. Grimsdale. Performance of Space Subdivision Techniques in Ray Tracing. *Computer Graphics forum*, 11(4):213–220, 1992.

[MT] Martin Mellado and Josep Tornero. On the Spherical Splines for Robot Modeling.

[MW88] Matthew Moore and Jane Wilhelms. Collision Detection and Response for Computer Animation. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 289–298, August 1988.

[NAB86] I. Navazo, and D. Ayala, and P. Brunet. A Geometric Modeller based on the Exact Octree Representation of Polyhedra. *Computer Graphics Forum*, 5(2):91–104, June 1986.

[NAT90] Bruce Naylor, and John Amanatides, and William Thibault. Merging BSP Trees Yields Polyhedral Set Operations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 115–124, August 1990.

[OB79] J. O'Rourke and N. Badler. Decomposition of Three-Dimensional Objects Into Spheres. *IEEE Trans. On Pattern Analysis and Machine Intelligence*, PAMI-1(3):295–305, 417, July 1979.

[PFTV88] William H. Press, Brian P. Flannery, and Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

[PY90] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[Reichling88] M. Reichling. On the detection of a common intersection of k convex polyhedra. In *Computational Geometry and its Applications*, volume 333 of *Lecture Notes in Computer Science*, pages 180–186. Springer-Verlag, 1988.

[Reischuk88] Rüdiger Reischuk. *Eeinführung in die Komplexitätestheorie.* B.G. Teubner, Stuttgart, 1988.

[RS92a] J. Ruppert and R. Seidel. On the difficulty of triangulating three-dimensional non-convex polyhedra. *Discrete Comput. Geom.*, 7:227–253, 1992.

[RS92b] J. Ruppert and R. Seidel. On the difficulty of triangulating three-dimensional non-convex polyhedra. *Discrete Comput. Geom.*, 7:227–253, 1992.

[Samet90a] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[Samet90b] Hanan Samet. *Applications of Spatial Data Structures.* Addison-Wesley, Reading, Massachusetts, 1990.

[Schönhardt28] E. Schönhardt. Über die Zerlegung von Dreieckspolyedern in Tetraeder. *Mathematische Annalen*, 98:309–312, 1928.

[SN86a] M. Szilvási-Nagy. Two Algorithms for Decomposing a Polyhedron into Convex Parts. *Computer Graphics Forum*, 5(3):197–202, September 1986.

[SN86b] M. Szilvási-Nagy. Two Algorithms for Decomposing a Polyhedron into Convex Parts. *Computer Graphics Forum*, 5(3):197–202, September 1986.

[Sung91] K. Sung. A DDA Octree Traversal Algorithm for Ray Tracing. In Werner Purgathofer, editor, *Eurographics '91*, pages 73–85. North-Holland, September 1991.

[SWF+93] John M. Snyder, Adam R. Woodbury, Kurt Fleischer, and Bena Currin, and Alan H. Barr. Interval Method for Multi-point Collision Between Time-dependent Curved Surfaces. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 321–334, August 1993.

[TKM84] M. Tamminen, and O. Karonen, and M. Mäntylä. Ray-Casting and Block Model Conversion Using a Spatial Index. *Computer Aided Design*, 16:203–208, July 1984.

[TN87] William C. Thibault and Bruce F. Naylor. Set Operations on Polyhedra Using Binary Space Partitioning Trees. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 153–162, July 1987.

[Torres90] Enric Torres. Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes. In C. E. Vandoni and D. A. Duce, editors, *Eurographics '90*, pages 507–518. North-Holland, September 1990.

[Toussaint88] Godfried T. Toussaint. Some collision avoidance problems in the plane. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40 of *NATO ASI*, pages 639–672. Springer-Verlag, 1988.

[TS84] M. Tamminen and H. Samet. Efficient octree conversion by connectivity labeling. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 43–51, July 1984.

[Vanecek Jr.91] G. Vanecek Jr. Brep-index: a multidimensional space partitioning tree. *Internat. J. Comput. Geom. Appl.*, 1(3):243–261, 1991.

[VMT94] Pascal Volino and Nadia Magnenat-Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. In M. Daehlen and L. Kjelldahl, editors, *Eurographics 1994*, number 3, pages C–155–C–166, Oslo, September 1994. Eurographics Association, Blackwell Publishers.

[Weiler85] K. Weiler. Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, January 1985.

[Welzl91] E. Welzl. Smallest enclosing disks, balls and ellipsoids. Report B 91-09, Fachbereich Mathematik, Freie Universität Berlin, Berlin, Germany, 1991.

[WHG84] Hank Weghorst, and Gary Hooper, and Donald P. Greenberg. Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[Woo85] T. C. Woo. A Combinatorial Analysis of Boundary Data Structure Schemata. *IEEE Comput. Graphics and Applications (USA)*, 5(3):19–27, March 1985.

[YDEP89] F. F. Yao, D. P. Dobkin, and H. Edelsbrunner, and M. S. Paterson. Partitioning space for range queries. *SIAM J. Comput.*, 18:371–384, 1989.

[YK86] Z. Yu and W. Khalil. *Table Look-Up for Collision Detection and Safe Operation of Robots*, pages 343–347. Pergamon Press, Vienna, Austria, December 1986.

[YKFT84] K. Yamaguchi, T. L. Kunii, and K. Fujimura, and H. Toriya. Octree-Related Data Structures and Algorithms. *IEEE Comput. Graphics and Appl. (USA)*, 3:53–59, January 1984.

[YW93] Ji-Hoon Youn and K. Wohn. Realtime Collision Detection for Virtual Reality Applications. In *IEEE Virtual Reality Annual International Symposium*, pages 415–421, September 18–22 1993.