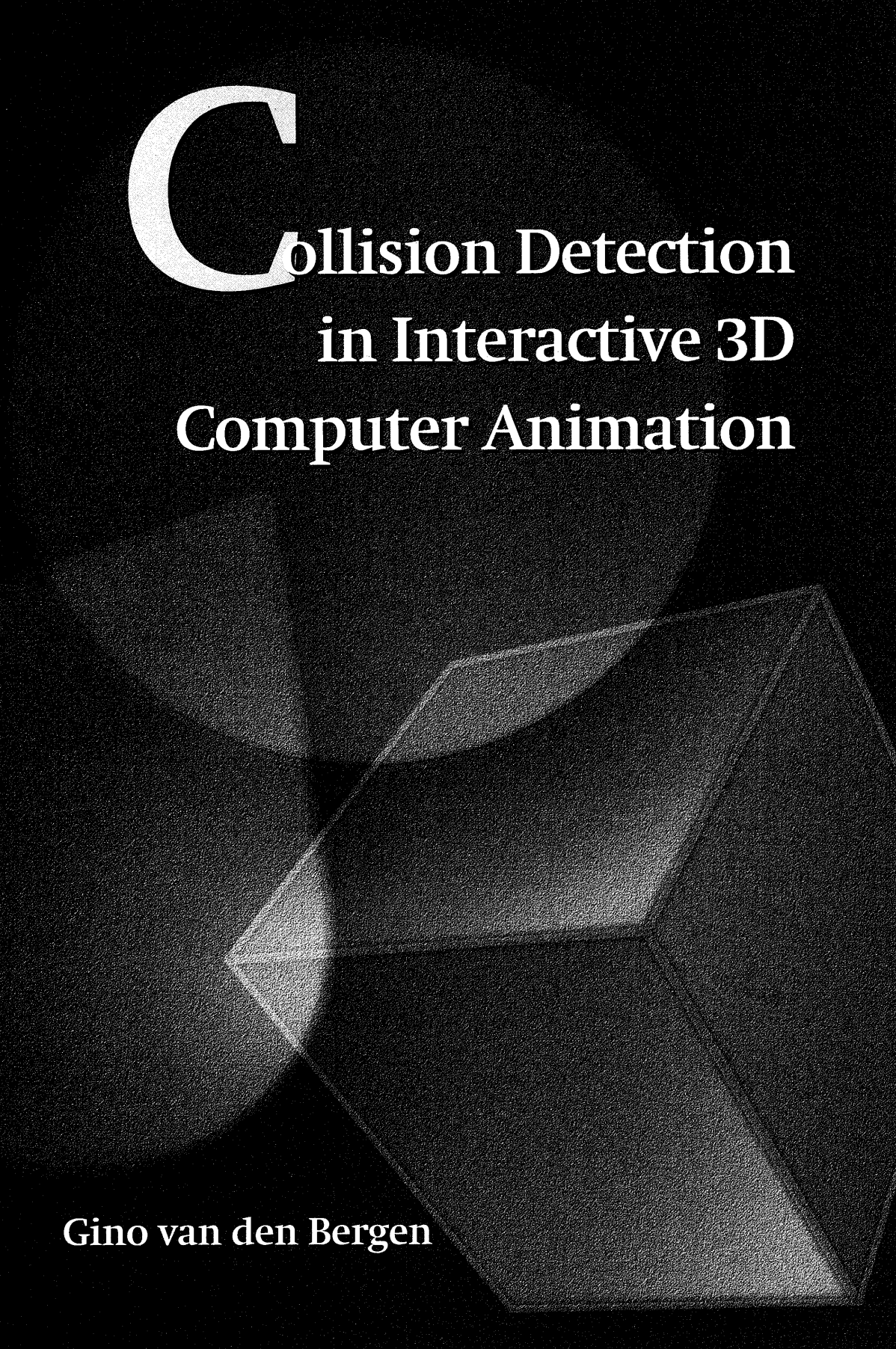
Collision Detection in Interactive 3D Computer Animation



Gino van den Bergen

Collision Detection in Interactive 3D Computer Animation

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op maandag 29 maart 1999 om 16.00 uur

door

GINO JOHANNES APOLONIA VAN DEN BERGEN

geboren te Hulst

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. D. K. Hammer
en
prof.dr. M. H. Overmars

Copromotor:

dr.ir. C.W.A.M. van Overveld

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Bergen, Gino van den

Collision detection in interactive 3D computer animation / Gino van den Bergen. - Eindhoven : Eindhoven University of Technology, 1999. Proefschrift.
ISBN 90-386-0671-0
NUGI 852
Subject headings: three-dimensional computer graphics / animation techniques / computational geometry
CR Subject Classification (1998) : I.3.5, I.3.7, F.2.2, G.1, G.4

Cover Design: Ben Mobach
Printed by University Press Facilities, Eindhoven

I made a big decision a little while ago.
I don't remember what it was, which prob'ly goes to show
That many times a simple choice can prove to be essential
Even though it often might appear inconsequential.

I must have been distracted when I left my home because
Left or right I'm sure I went. (I wonder which it was!)
Anyway, I never veered: I walked in that direction
Utterly absorbed, it seems, in quiet introspection.

For no reason I can think of, I've wandered far astray
And that is how I got to where I find myself today.

Contents

Chapter 1

Introduction

"One never notices what has been done.
One can only see what remains to be done."
Marie Curie

Current state-of-the-art in computer graphics enables us to interactively explore three-dimensional data, such as architecture and scientific visualizations. In many applications, these data represent environments with behavior, for instance, in games and simulators. Often, the goal of such applications is to simulate some aspects of the real world as accurately as possible. A term often used for this type of applications is **virtual reality**, although this term typically refers to immersively experienced environments that utilize a head-mounted display and a 3D pointing and grabbing device such as a data glove.

One aspect of the real world that greatly affects the manner in which we experience an environment is the constraint that at the same time two material objects cannot occupy the same point in space (at low energies [51]). Occasionally, we regard this constraint as undesired, since it restricts our motions. However, impenetrability enables manipulations, such as pushing and stacking objects. Also the fact that we can stand and walk depends on the ground being impenetrable.

In general, object representations in simulated environments do not impose impenetrability. If we want a simulated environment to behave according to the real world with respect to the impenetrability of material objects, we need to incorporate a mechanism that enforces this constraint. An important part of such a mechanism is detecting configurations of interpenetrating objects, which are called collisions.

We refer to the agent responsible for resolving collisions as the **collision handler**. The collision handler often needs additional data pertaining

to the configuration of the colliding objects. These data are called **response data**. For example, in physics-based simulations, collisions are resolved by simulating the effect of the forces that act on a pair of colliding objects as a result of their impenetrability. Here, the response data are a contact point and a contact plane at the moment of first contact.

1.1 Problem Domain

In this thesis, we address the problem of detecting collisions among moving 3D objects. In particular, we focus on collision detection for application in interactive 3D computer animation. We restrict ourselves to shape representation types that are commonly used in interactive 3D graphics, and pay special attention to the performance of the collision detection algorithms.

In computer animation, object configurations are given only for discrete time steps, called frames. Collision detection is performed only for these time steps. No prior knowledge concerning the trajectories, motions, and velocities of the objects is assumed. However, we often assume some degree of coherence between the configurations of objects in two consecutive frames.

In interactive animation, each frame is computed in real-time. In order for the human eye to experience smooth motion, the animation system needs to display at least 25 frames per second. Hence, all tasks required for computing a single frame, including detecting and resolving collisions, should take no more than a few hundredths of a second in total. For this reason, the performance of the used collision detection methods is crucial, and thus receives our main attention. However, we will also address other attributes, such as usage of storage space, and preprocessing time.

We will consider collision detection algorithms for objects represented by shapes composed of primitive shapes, such as polygons, convex polyhedra, spheres, boxes, cones, and cylinders. Other methods for describing 3D objects, such as constructive solid geometry (CSG) and implicit surfaces, are not (yet) applied in interactive 3D graphics, and are not considered here.

Besides collision detection methods, we also examine methods for computing response data for the mentioned shape types. The response data types for colliding objects that we consider are: a common point, i.e., a point that is contained in both objects, a representation of the intersection, and an approximation of a contact point and a contact plane.

1.2 Historical Background

The earliest applications of 3D collision detection are found in robotics and automation [10]. Here, product assembly or test facilities are simulated on a computer in order to verify interference problems. The different objects to be checked for interference are represented as polyhedra. Interference checking in robotics simulations is often performed on a continuous rather than a discrete time axis, i.e., the objects are checked for interference in continuous four-dimensional space-time [11, 12, 15]. However, this approach is applicable only for a limited class of objects and motions.

The problem of computing the minimum distance between two objects is more general than the problem of finding an intersection. Hence, algorithms for computing the distance are useful for collision detection as well. Two early algorithms for computing the distance between a pair of convex polyhedra are described in [14] and [40]. Moreover, in combination with the velocities of the objects, the distance between the objects allows us to estimate the collision times [24], which is useful in robotics simulations, where the bounds of the velocities are a priori known.

The first uses of collision detection in 3D computer animation are found in physics-based simulations [68, 49, 2]. Although his work is not directly aimed at interactive applications, Baraff was the first to exploit coherence in-between frames in order to improve the performance of the collision detection [3]. Baraff exploits frame coherence in two ways. He uses a scheme that allows updating the list of objects pairs for which the bounding boxes overlap in time that is expected linear in the number of objects, when coherence is high. Furthermore, Baraff caches separating planes of convex object pairs that are found to be disjoint. These separating planes are used for quickly answering intersection queries in the next frame.

A similar technique is used in an algorithm by Lin and Canny for computing the distance between convex polyhedra [61]. Here, the closest features (vertices, edges, facets) of a pair of polyhedra are cached, and incrementally updated in each new frame. An update of the closest feature pair takes roughly constant time when frame coherence is high. The Lin-Canny algorithm is applied in I-COLLIDE, which is the first collision detection library for interactive applications to become publicly available [22].

After Lin-Canny, other incremental algorithms for convex polyhedra followed that have the same time bound. Chung and Wang presented an algorithm for updating a separating axis of a pair of disjoint polyhedra, similar to Baraff's separating planes [20]. Cameron presented in [13], an enhanced version of the Gilbert-Johnson-Keerthi distance algorithm [40],

which is used for incrementally computing the distance between a pair of polyhedra. Finally, Mirtich presented an enhanced version of the Lin-Canny algorithm, which is claimed to have improved performance and robustness [66].

Current state-of-the-art in interactive 3D graphics allows the use of shapes composed of thousands of primitives. In order to reduce the number of pairwise primitive intersection tests in collision detection of objects represented by such shapes, spatial data structures are often applied. These data structures are used to quickly reject a large number of primitives from intersection testing, based on their geometric location. In the last few years, spatial data structures that are used for this purpose have received a lot of attention.

Data structures based on space partitioning techniques were explored earliest. Garcia-Alonso, Serrano, and Flaquer presented a solution based on voxel grids [38]. Zachmann and Felger presented a data structure based on the k-d tree, called *BoxTree* [106]. Model partitioning techniques incorporating bounding volume hierarchies are currently the mostly used. Palmer and Grimsdale [78] and Hubbard [56] apply hierarchies of spheres for speeding up collision detection. Gottschalk e.a. presented a structure called *OBBTree*, which uses oriented bounding boxes as volumes [46]. An implementation of the algorithms for constructing and testing OBBTrees has been made publicly available in form of a collision detection package called RAPID [44]. Recently, both Klosowski e.a. and Zachmann presented hierarchical structures of discrete-orientation polytopes for collision detection [59, 105]. All these structures are static, and are thus applicable only to rigid objects.

Collision detection methods for deformable polygonal objects have not yet been extensively explored. Work by Smith e.a. describes a dynamic data structure based on the octree for speeding up collision detection of deformable objects [88]. Their algorithm runs in a time that is close to being linear in the total number of polygons, which is much slower than the algorithms for rigid objects. Fast algorithms for deformable models are still an important research topic.

Other challenges that remain are improving the robustness and performance of object intersection tests. We see that in the last few years, most of the innovations in 3D collision detection are aimed at improving these two qualities, however, further research is still necessary.

Finally, with the adoption of multiple shape representation methods in 3D modeling for computer animation, as for instance in VRML [8], the need arises for collision detection algorithms that can be applied to collections of objects that are composed of a mix of shape types. The question

of how to efficiently deal with multiple shape types in collision detection still needs to be addressed.

1.3 Organization and Contributions

The rest of this thesis is organized as follows. In Chapter 2, we define the concepts used in this thesis. We discuss different types of shape representations, different types of motion, response data types, and efficiency considerations.

In Chapter 3, we discuss a number of algorithms for testing intersections between non-convex polygons, and between a polygon and a number of volume types. Here, we present new algorithms for testing and computing the intersection of a pair of non-convex polygons, which run in time linear in the number vertices of the polygons.

Chapter 4 describes algorithms for collision detection of convex objects, mostly algorithms for convex polyhedra. We discuss algorithms for finding a common point, for finding a separating axis, and for computing the distance. In particular, we look into incremental algorithms that exploit frame coherence. We present a new algorithm for incrementally computing a separating axis. The algorithm is based on the Gilbert-Johnson-Keerthi algorithm, and is called ISA-GJK. The ISA-GJK algorithm has the following properties:

- ISA-GJK is, although slightly slower than the Chung-Wang algorithm, significantly faster than incremental distance algorithms, such as Lin-Canny.
- ISA-GJK is applicable to convex objects in general, not merely to convex polyhedra, as for instance Lin-Canny and Chung-Wang.
- ISA-GJK does not have termination problems as the original GJK has for degenerate configurations of objects.

Hence, ISA-GJK's strong point is versatility, whereas, of the mentioned algorithms for collision detection of convex objects, its performance is surpassed only by Chung-Wang's.

In Chapter 5 we discuss spatial data structures that are used for speeding up collision detection of models composed of a large number of objects. We present an improvement of Baraff's incremental sweep and prune scheme [4] for maintaining a list of pairs of objects whose world-axes aligned bounding volumes overlap. Our version allows update times

that are roughly linear in the number of moving objects only, rather than the total number of objects, when frame coherence is high. Furthermore, we show how a data structure, called AABB tree, can be used for speeding up intersection testing between two complex deformable shapes. In comparison to the OBB tree, a comparable data structure, we found the following:

- Although AABB trees are not as fast as OBB trees, they are not much slower, even in cases where the degree of overlap between the shapes is high. In many cases, intersection tests of a pairs of AABB trees take less than 50% more time than intersection tests on OBB trees for the same shapes.

- AABB trees take roughly half as much storage as an OBB tree, and take less time to construct.

- AABB trees can be updated reasonably fast after shape deformations. Updating an OBB tree is considerably more complex.

Hence, we found the AABB tree, as presented in this thesis, to be the data structure of choice for intersection tests of complex deformable shapes. Furthermore, since they are not much slower than OBB trees, AABB trees are a reasonable choice for rigid objects as well.

In Chapter 6 we describe the design of SOLID, a collision detection library for interactive 3D computer animation. SOLID incorporates the following innovative features:

- SOLID supports models composed of a mix of shape types, including boxes, cones, cylinders, spheres, simplices, convex polygons, and convex polyhedra.

- SOLID supports deformations of complex shapes.

- SOLID allows besides translations and rotations, also nonuniform scalings on objects.

- SOLID optionally computes response data that represent the approximated contact points and contact plane of a pair of colliding objects.

The complete source code and documentation of SOLID have been made publicly available under the terms of the GNU Library General Public License [35]. Details on how to obtain the source code can be found at URL: *http://www.win.tue.nl/cs/tt/gino/solid.*

Finally, Chapter 7 summarizes the results of our work, and presents some pointers to interesting topics for future work.

A brief explanation of the notational conventions, as well as some linear analysis concepts can be found in Appendix A. Furthermore, we include a reference guide for SOLID version 2.0 in Appendix B.

Chapter 2

Concepts

"Don't reinvent the wheel. Just realign it."
Anthony J. D'Angelo

In this chapter we define the concepts that are relevant within the context of this thesis. We discuss a number of commonly used methods for representing objects. We also describe different types of motion used in computer animation, and discuss the problems of sampled motion. Furthermore, we will look into the computation of response data for physics-based simulations. Finally, we will cover some efficiency considerations, such as coherence and storage, and discuss the difficulties in measuring performance.

2.1 Objects

In this section we define the class of objects for which collision detection algorithms are presented in this thesis. An **object** is a closed bounded nonempty set of points in three-dimensional Euclidean space. The **dimension** of an object is the dimension of its affine hull. An object is **convex** if it contains all the line segments connecting any pair of its points. Convex objects often allow simpler or faster algorithms for intersection testing. In Chapter 4, we will discuss a number of algorithms that are applicable for collision detection of convex objects only.

Objects may be composed of simpler objects called **primitive shapes**, or **primitives** for short. Primitives are the building blocks of the objects in the simulated environment. The primitives we consider are the common primitives for geometric modeling: spheres, cones, cylinders, boxes, and polygons, as used for instance in VRML [8]. Furthermore, polytopes

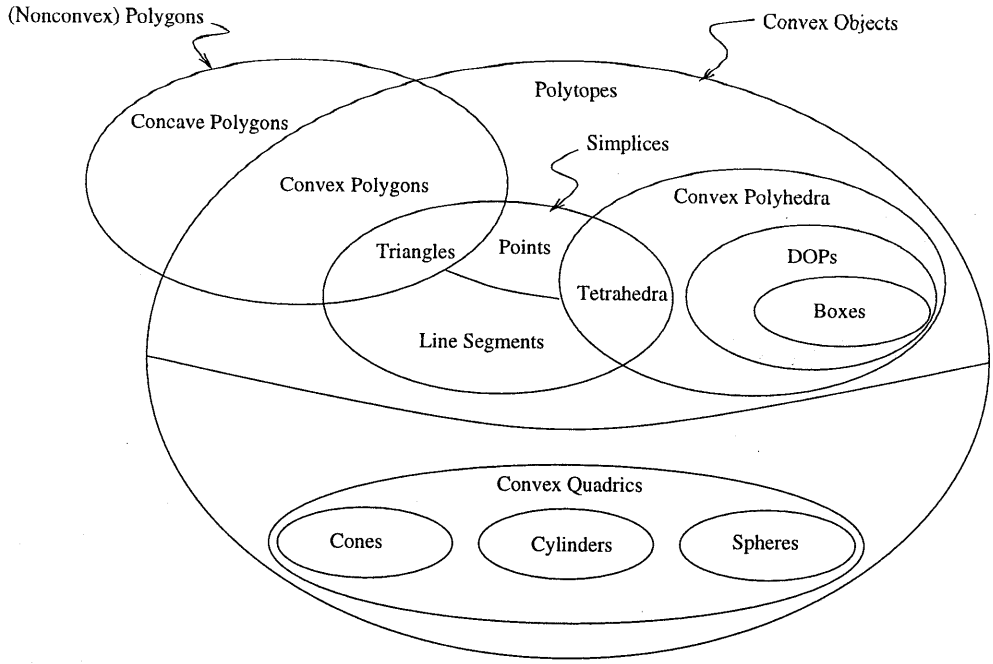


Figure 2.1: A taxonomy of primitive types

are also considered primitives. A precise definition of the term **polytope** is presented further on. For now, let us define a polytope as a convex object whose boundary is composed of a finite number of flat facets. Figure 2.1 shows a taxonomy of the types of primitives we consider. Here, the term DOP denotes *discrete-orientation polytope*, i.e., a three-dimensional polytope, whose facet orientations are chosen from a fixed finite set of orientations.

Concave polyhedra are not considered primitives. They are described as a grouping of primitives, either by decomposition into convex parts, or by boundary representation as a set of polygons. We will discuss the merits of both representations in Chapter 3. Let us have a closer look at the different primitive types now.

2.1.1 Polygons

Polygons are currently the most commonly used modeling primitives in 3D graphics. A **polygon** is the region of a plane bounded by a closed chain of line segments that lie in the plane. Let $\mathbf{p}_0, \ldots, \mathbf{p}_{n-1}$ be coplanar points. For $i = 0, \ldots, n-1$, the i-th line segment in the boundary of the polygon

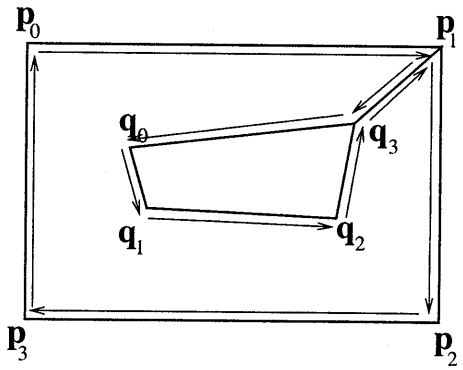


Figure 2.2: Fixing a hole in a polygon

defined by this sequence of points is the segment connecting $\mathbf{p}_i$ and $\mathbf{p}_{i\oplus 1}$, where $\oplus$ denotes addition modulo n. The points are referred to as **vertices** and the segments as **edges**. A polygon is called **simple** if no two edges intersect, other than the edges that share a vertex.

According to this definition, a simple polygon does not have holes. However, we allow a pair of identical edges in a polygon's boundary such that a polygons with holes may be constructed in the following way. Let $\mathbf{p}_0, \ldots, \mathbf{p}_{m-1}$ be a polygon's outer boundary in clockwise orientation, and $\mathbf{q}_0, \ldots, \mathbf{q}_{n-1}$, the boundary of a hole in counter-clockwise orientation, and let $\mathbf{p}_i$ and $\mathbf{q}_j$ be the closest pair of the two chains. Then, this polygon is represented as a single chain of edges defined by the list

$$\mathbf{p}_0, \ldots, \mathbf{p}_i, \mathbf{q}_j, \ldots, \mathbf{q}_{n-1}, \mathbf{q}_0, \ldots, \mathbf{q}_j, \mathbf{p}_i, \ldots, \mathbf{p}_{m-1}.$$

By connecting the two chains at the closest points we avoid constructing a non-simple polygon, since the edges connecting these points can not cross any of the other edges. Figure 2.2 illustrates this construction. We see that the polygon has a pair of identical oppositely directed edges.

For most applications, in particular for visualization, a representation of a polygon as a list of vertices suffices. However, sometimes a representation of the supporting plane of a polygon is convenient. The plane supporting a triangle is computed by simply taking the cross product of two of its edges as normal, and one of the vertices as point in the plane. For polygons with more than three vertices, we may select three non-collinear vertices, and compute the plane for the triangle formed by these three vertices. However, since polygon vertices represented by machine numbers are often not exactly coplanar, the plane may deviate considerably from

the best-fit plane through the vertices if the three vertices are ill-chosen. A better alternative is Newell's method [93, 94], which produces a plane that approximates the best-fit plane more closely.

For convex polygons, we can use simpler and faster rendering routines than for non-convex polygons [1]. As a result of this, graphics libraries and hardware usually support convex polygons only [102]. Hence, the majority of polygonal models that are used in 3D graphics applications are composed of convex polygons.

Conveniently, convex polygons also allow faster algorithms for intersection testing, as we will see in Chapter 3 and 4. In order to use these algorithms for models composed of non-convex polygons, it is necessary to decompose concave polygons into convex subparts, for instance by triangulation. Algorithms for triangulating non-convex polygons can be found in [77].

2.1.2 Polytopes

A **polytope** is the convex hull of a finite point set. The **convex hull** of a finite point set $A = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$ is the set of **convex combinations** of points in A, defined by

$$\text{conv}(A) = \left\{ \sum_{i=1}^{n} \lambda_i \mathbf{a}_i : \sum_{i=1}^{n} \lambda_i = 1, \lambda_i \geq 0 \right\}.$$

The set of **vertices** of a polytope $P = \text{conv}(X)$, denoted by $\text{vert}(P)$, is the smallest set $Y \subseteq X$, such that $\text{conv}(Y) = P$. A **simplex** is the convex hull of an affinely independent set of points. Simplices of one, two, three, and four vertices are respectively points, line segments, triangles, and tetrahedra. The dimension of a polytope is the dimension of its affine hull. The set of two- and three-dimensional polytopes is respectively the set of convex polygons and the set of convex polyhedra.

In most cases, we will use a list of vertices as a basic representation for polytopes. However, a polytope may also be represented as the intersection of a finite set of closed halfspaces. In fact, any object that is a bounded intersection of a finite set of closed halfspaces is a polytope [48]. According to *Euler's formula*, the minimum number of halfspaces that are required for representing a given polytope is roughly linear in the number of vertices of the polytope [80].

In some applications, representing a polytope by a collection of halfspaces is more convenient than using a vertex representation. For instance,

polytope types that are applied as bounding volumes, such as discrete-orientation polytopes, rely on a halfspace representation. A **discrete-orientation polytope** (DOP) is the intersection of a fixed number of slabs. A **slab** is the intersection of a pair of oppositely oriented halfspaces, i.e., a region of space bounded by a pair of parallel planes. Each slab is oriented according to a fixed axis relative to the objects coordinate system. For each DOP we use the same set of axes, hence, the description of the axes is not part of a DOP's representation.

A k-DOP is the intersection of k slabs.[1] For $\mathbf{d}_1, \ldots, \mathbf{d}_k$, a set of axes, we define the k-DOP represented by extents β_i and η_i, as the point set

$$\{\mathbf{x} \in \mathbb{R}^3 : \beta_i \leq \mathbf{d}_i \cdot \mathbf{x} \leq \eta_i, \text{ for } i = 1, \ldots, k\}.$$

Due to their small storage requirements ($2k$ scalars), k-DOPs are well-suited for use as a bounding volume.

A 3-DOP is better known as a **parallelepiped**, or simply **box**. Boxes may alternatively be represented by a center point $\mathbf{c}$ and extent vector η_i, defining the point set

$$\{\mathbf{x} \in \mathbb{R}^3 : |\mathbf{d}_i \cdot (\mathbf{x} - \mathbf{c})| \leq \eta_i, \text{ for } i = 1, 2, 3\},$$

which is a more convenient representation for some operations.

A **feature**[2] of a polytope is the intersection of the polytope with a supporting plane. The features of zero, one, and two dimensions are called vertices, edges, and facets, respectively. A **boundary representation** of a polytope is the set of its features together with its incidence relation. The boundary representation of a polygon is simply the chain of its edges. A polyhedron's boundary representation has a planar-graph topology. We will describe a number of data structures that can be used to represent the boundary of a polyhedron. For a more thorough discussion of the representation of polyhedron boundaries, the reader is referred to [57].

The best-known data structure for representing the boundary of polyhedron is Baumgart's **winged-edge structure** [7]. Often, we do not require a complete representation of the boundary, but are only interested in the adjacency graph of the polytope's vertices. In these cases, we may use a simplified variant of the winged-edge structure, in which the references to the adjacent facets of each edge are removed. This structure is better known as **doubly-connected edge list** (DCEL) [80].

[1] In contrast with [59], we count the number of slabs rather than the number of halfspaces.

[2] In geometry literature the common term is **face**. However, we avoid using this term, since in computer graphics texts, face is a synonym for polygon.

In winged-edge-type structures, a node represents an undirected edge in the vertex adjacency graph. This leads to inefficient case distinction in graph traversals, since it is necessary to determine the orientation of each edge traversed in order to find the proper successor. This problem is solved in the **halfedge structure**, in which each undirected edge is represented by a pair of directed edges [57].

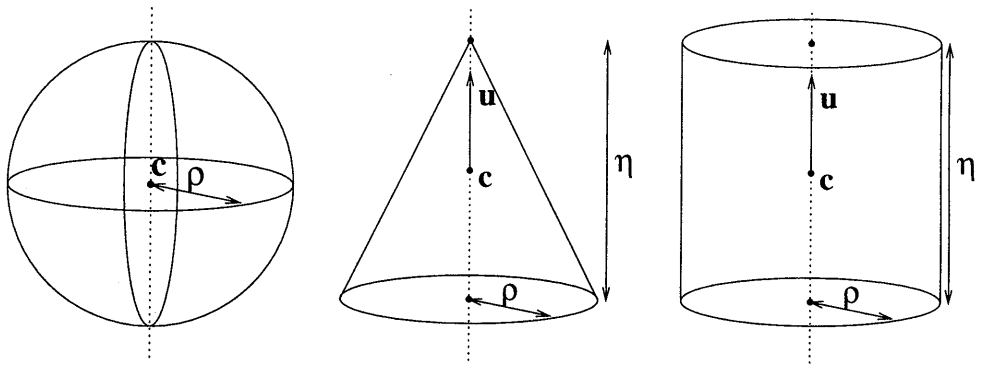
In computer animation, the boundaries of polyhedra are often topologically invariant. Hence, we do not rely on efficient operations for modifying a boundary representation. In these cases, the simplest representation of the vertex adjacency graph, is often the best. For each vertex, we maintain a list (array) of pointers (indices) to its neighboring vertices. In this way, an edge is represented by two list entries corresponding to the edges endpoints. Hence, this representation requires storage that is linear in the number of edges, as is the case for the other boundary representation structures we have mentioned.

2.1.3 Quadrics

A **quadric** is an object that has quadratic surface elements. Quadrics are considered solids rather than surfaces, i.e., the interior is part of the object. We consider convex quadrics, such as spheres, capped cones, and capped cylinders, as primitives. Although for interactive visualization, these shapes are often represented by convex polyhedra, it is possible, and usually more efficient and accurate to use their exact quadric representation for intersection testing, as we will discover in Chapter 4.

A **sphere** is represented by a center point $\mathbf{c}$ and a radius ρ. A **cone** is represented by a center point $\mathbf{c}$ (halfway between the apex and the base), a unit vector $\mathbf{u}$ that spans its central axis, and two positive scalars η and ρ which are its height and its radius at the base. A **cylinder** is represented by a center point $\mathbf{c}$, a unit vector $\mathbf{u}$ that spans its central axis, and and two positive scalars η and ρ which are its height and radius. See Figure 2.3 for a visual description of these primitives.

In VRML, the center point and central axis of the quadric and box primitives is fixed and cannot be used as a shape parameter [8]. The local origin and y-axis are taken to be, respectively, the center point and central axis of the primitives. This convention does not restrict the set of objects that can be specified using the language, since the primitives may be placed at arbitrary positions and orientations by applying affine transformations. In the following subsection, we further discuss the use of affine transformations for building models.



(a) Sphere (b) Cone (c) Cylinder

Figure 2.3: The three quadric primitives

2.1.4 The Scene Graph

So far, we discussed the different types of primitives that are commonly used for building models. Here, we will examine how these shapes are combined to create complex models.

Complex models are constructed in the first place by grouping primitives. Several of these groupings may in turn be grouped to form higher-level groupings. We thus get a hierarchical description of a model. The object represented by a grouping of sub-objects is the union of the sub-objects. Note that, contrary to CSG representations, we do not allow intersections and set differences as operations on the objects in a grouping.

Furthermore, we allow objects in a model to be defined in their own local coordinate system relative to the coordinate system of the model. A coordinate system is defined by the directions and scales of the three axes and the position of the origin. The common way to represent a local coordinate system relative to a reference coordinate system is as an affine transformation. An **affine transformation** is a mapping of the form $\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$, where $\mathbf{B}$ is a 3×3 matrix, defining the linear component, and $\mathbf{c}$ the translation of the transformation. The columns of $\mathbf{B}$ are the vectors spanning the local coordinate axes relative to the reference coordinate system, and the point $\mathbf{c}$ is the position of the local origin in reference coordinates. It can be seen that, for a point $\mathbf{x}$ in local coordinates, $\mathbf{T}(\mathbf{x})$ will give us the reference coordinates of the point. See Appendix A for details on affine transformations and coordinate systems.

Affine transformations provide us with a powerful mechanism for creating models. We may define a shape in its local coordinate system and use it to create multiple objects, each object being the result of a different placement of the shape's local coordinate system. In this way, we may use the same shape to create objects at different positions, orientations and scalings.

The structure we get using these construction methods is called a **scene graph**. A scene graph is a directed acyclic graph (DAG), i.e., a tree-like structure in which a single node may have multiple parents. The structure has two types of internal nodes: grouping nodes and transform nodes. A **grouping node** simply combines a number of sub-graphs. A **transform node** defines a new local coordinate system for its descendants relative the current coordinate system. The coordinate system corresponding to the root of a scene graph is the **world coordinate system**. Hence, we find the local coordinate system of a primitive in the graph relative to the world coordinate system to be the concatenation of the transformations along the root path of the primitive.

2.2 Animation

Animation adds a time parameter to the object representation, i.e., the set of points that comprise an object represented by a node in the scene graph is a function of time. We refer to the object represented by a node at a given instance of time t as the **configuration** of the node at time t.

We formally define the term **collision** as a state at an instance of time in which the configurations of two nodes intersect. This definition is meaningful only for nodes that lie on different root paths, since otherwise the configurations would trivially intersect at any time. Here, two objects A and B **intersect** if $A \cap B \neq \emptyset$. Hence, since objects are closed, a pair of objects in contact are considered intersecting.

Next, we will discuss the different types of motion that may be used to animate a model.

2.2.1 Motion

Animation is created by making some or all the components of a model time-dependent. We concentrate on model changes over time, that affect the geometric attributes only. We discern two types of motion corresponding to the different node types in a scene graph:

1. The most common way to animate a model is by altering the transformations in the transform nodes of the scene graph. We will refer to this type of motion as change of **placement**. The placement changes are the result of translations, rotations, and nonuniform scalings on (part of) the model. Most commonly used are rigid motions, i.e., transformations that use translations and rotations only. In kinematics, the rigid motions of parts of the model are usually constrained in one or more degrees of freedom in order to create joints. For instance, the knees of a walking figure have one rotational degree of freedom. Models that are animated by motion of joints are referred to as **articulated models**.

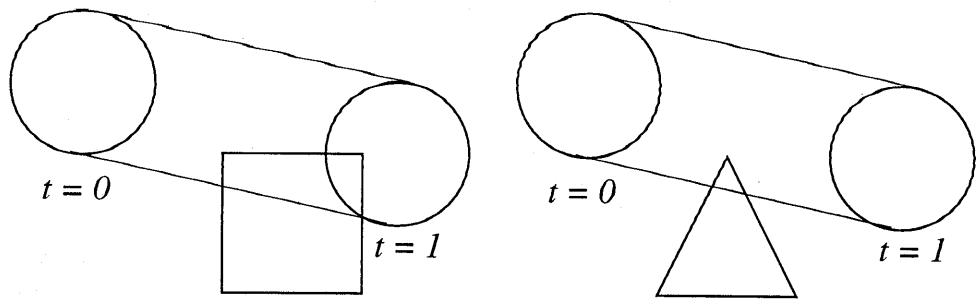
2. Animation may also be created by changing the geometries of the primitive shapes. This type of motion is called **deformation**. This is most commonly done for objects represented by polytopes or polygon meshes. Here, the positions of the vertices of the polytopes or polygons are time-dependent. Deformations may be applied to simulate for instance fluids, cloths, or skin.

2.2.2 Time

In the real world time is assumed to be continuous, i.e., the time interval in-between two consecutive events can be arbitrary short. This suggests that we need to solve the collision detection problem as an intersection test in continuous four-dimensional space (space-time). However, such an approach leads to practical difficulties.

Solutions to the four-dimensional intersection detection problem have been presented for a restricted class of objects and motions [11, 12, 15, 81]. However, in many common cases, the four-dimensional object that is swept by a three-dimensional object in motion is often too complicated to make a four-dimensional intersection test computationally feasible at interactive rates. For instance, it is hard to test the screw-like volume swept by a spinning object, such as a propellor or a fan, for intersection with another moving object's swept volume.

Hence, for interactive simulations, we opt for a motion description given by discretely sampled configurations at fixed time intervals. These configurations of a model at discrete time steps are referred to as **frames**, as used in the context of computer animation. Models are tested for interference for these fixed frames only. Collisions that occur in-between two discrete time steps are therefore not detected. Testing intersections for discrete frames only may result in collisions being detected too late or not at



(a) Too late (b) Missed

Figure 2.4: Problems when detecting collisions at discrete time steps

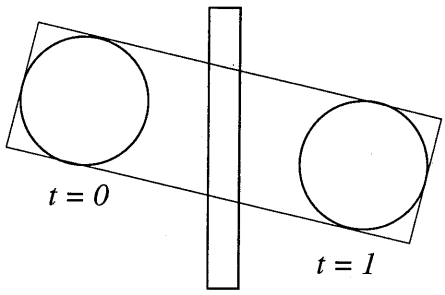
all, as illustrated in Figure 2.4.

Missing collision may result in undesired behavior, such as high-speed objects traversing through obstacles. For instance, a fired bullet may pass through a wall without colliding with the wall. We may deal with this problem by encapsulating two consecutive configurations of a model by a bounding volume, thus approximating the volume swept by the model over one time step. By testing these bounding volumes for intersection with obstacles, we can detect high-speed objects passing through these obstacles. Another solution is to stretch the model by a nonuniform scaling along the direction of the velocity, such that two consecutive configurations of the model overlap. See Figure 2.5 for an illustration of these solutions.

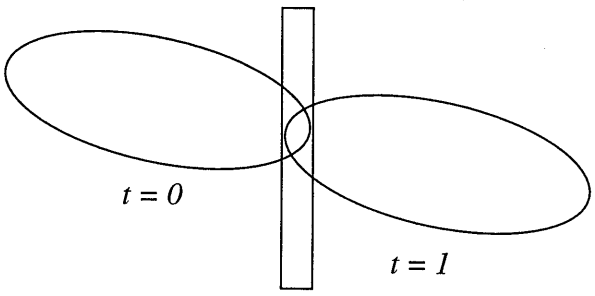
2.3 Response

Obviously, the reason for performing collision detection is to have some type of response to a collision. For some forms of collision response, no additional data concerning the colliding objects are needed. For instance, most flight simulators respond to a plane crash by ending the simulation, and displaying a message.

However, for most forms of collision response, additional data are used for response computations. For instance, in VR applications the exact spot where the simulated hand of the operator touches an objects is often required for collision response. Here, a common point of the object and the hand is the collision data.



(a) Encapsulation



(b) Stretching

Figure 2.5: Solutions to avoid missing collisions

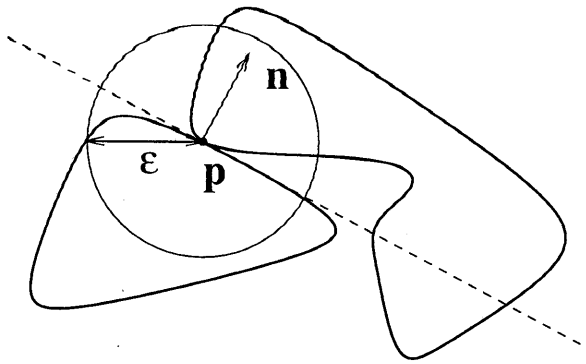


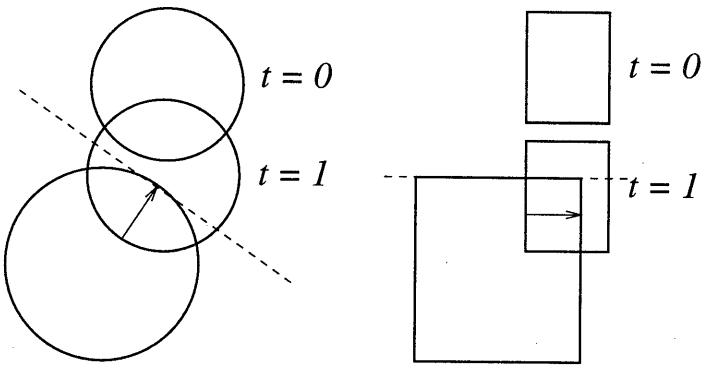
Figure 2.6: A contact plane defined by a contact point **p** and normal **n**

In the application of collision detection to physics-based simulations, it is necessary to have (an approximation of) a contact plane and a contact point for a colliding pair of objects in order to compute the reaction forces that resolve the collision. A **contact point** is a point where the objects first touch, and a **contact plane** is a plane that passes through the contact point and is oriented such that the intersections of the objects with an ε-neighborhood of any contact point lie on different sides of the plane, as depicted in Figure 2.6. It is assumed that by choosing ε small enough, each object's intersection with the ε-neighborhood can be regarded as a convex set, hence, a contact plane always exists. Note that neither a contact point nor a contact plane are necessarily unique.

Since collision detection is performed for discrete frames only, a pair of colliding objects will overlap each other to some extent at the moment that a collision is detected. From a configuration of intersecting objects it is rather difficult to estimate a contact plane and a contact point.

We may approximate the orientation of the plane using the minimum translational distance (MTD) between the objects [14], which is the length of the shortest translation that results in the objects being in contact. The direction of this translation can be used as an approximation of the contact plane's normal. However, for some configurations we may get undesired results, as shown in Figure 2.7. Here, the minimum translation is orthogonal to the normal of the actual contact plane for the rectangle example.

Better results are achieved by using the previous frame, i.e., the frame prior to the collision, for estimating a contact plane. For this purpose, a pair of closest points of the objects in the previous frame are determined.



(a) Fairly accurate (b) Inaccurate

Figure 2.7: Using the minimum translational distance for approximating a contact plane

A pair of closest points is pair of points, one from each object, such that their distance is the shortest of all point pairs. We use the difference of the closest points as the normal of the contact plane, which is a fairly accurate approximation of an actual contact plane's normal, as depicted in Figure 2.8. The closest points may be used as the points of the objects on which the reaction forces are applied. The reaction forces are directed along the normal of the contact plane, which is determined by the vector difference of the closest points. A discussion of methods used for physics-based simulations falls outside the scope of this thesis. The reader is referred to [6] for a thorough treatment of techniques used for interactive dynamics simulations.

2.4 Efficiency

In interactive computer animation, the available computational time per frame for collision detection is constrained by the desired frame rate. In order to experience real-time response, the frame rate needs to be at least in the order of 25 frames per second. We see that this constraint imposes quite strict demands upon the performance of the used collision detection methods.

Often, we observe a trade-off of computational cost against storage usage, for instance, when preprocessing or caching is done. Although in

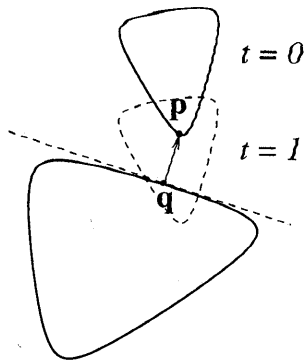


Figure 2.8: For a pair of closest points **p** and **q** in the frame prior to a collision, the difference $\mathbf{p} - \mathbf{q}$ is an accurate approximation of a contact plane's orientation.

many cases, usage of space is less critical than usage of time, the amount of available storage space is nevertheless limited. Therefore, we also have to take the amount of used storage space into account in choosing a collision detection method. In this section, we discuss a number of efficiency considerations that may govern the choice of algorithm used for collision detection.

2.4.1 Frame Coherence

Under the assumption that the changes per frame are small, i.e. the motion is smooth, the computations for detecting collisions are repeated for mostly the same input values. By caching and reuse of earlier computations the computation time per frame may be greatly reduced. The measure of reusability of computations from earlier frames is called **frame coherence**.

Witnesses play an important role in the exploitation of frame coherence. A **witness** is some piece of data pertaining to the current configuration of a pair of objects, that can be used for quickly answering future intersection queries on the objects, provided that the configuration does not change much. A witness may be either positive, i.e., the objects intersect, or negative, i.e., the objects are disjoint. An example of a positive witness is a common point of both objects. For convex objects, we may use a **separating plane**, or a **separating axis**, i.e., an axis orthogonal to a separating plane, as a negative witness. A closest point pair may be used

both as positive and negative witness. We will discuss the computation of these witnesses for convex objects in Chapter 4.

The use of cached witnesses is based on the idea that testing whether a witness from a previous frame is still valid in the current frame is cheaper than repeating the witness computation from scratch for the current frame. For instance, a point containment test is for many object types cheaper than an object intersection test. Therefore, if a witness from a previous frame is likely to be a witness in the current frame, as a result of a high degree of frame coherence, we may save ourselves some time by first testing the validity of the cached witness in the current frame. Only if the witness test fails, then an expensive intersection test needs to be performed, which may result in a new witness being computed.

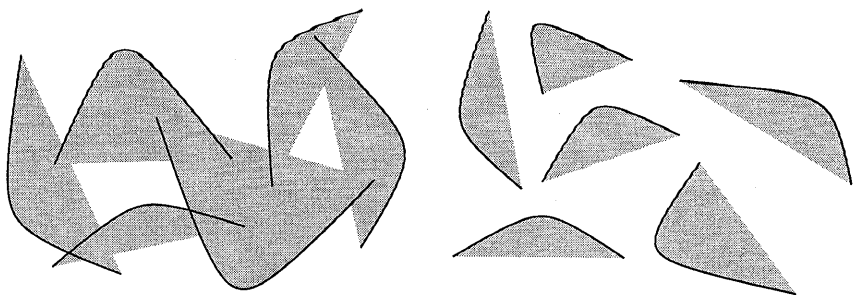
Besides the cost of testing their validity, some additional overhead cost are involved when using witnesses. The witnesses need to be cached in a data structure, and retrieved in the following frames, which obviously takes some time. Therefore, it is wise to cache witnesses only if they have a high probability of being valid witnesses in following frames. In computer animation, collisions are usually resolved rather than maintained. Hence, in this context, negative witnesses are more useful than positive witnesses, since most object pairs will be disjoint most of the time.

2.4.2 Geometric Coherence

Another type of coherence, which we will refer to as **geometric coherence**, may also be exploited for improving the performance of collision detection. Geometric coherence is the quality of a complex model that expresses the degree in which the objects in the model can be ordered geometrically, i.e., according to the regions of space that they occupy. It is hard to give a formal definition of the notion of geometric coherence, since the notion is strongly related to the method of ordering that is applied. However, we shall try to give a formal definition that is more or less independent of a specific ordering method.

Geometric coherence of a complex model is best described as the degree of *separability* of the set of objects in the model. Two objects are separable if the regions occupied by the objects, defined by their convex hulls, are disjoint. The degree of separability decreases if the degree of overlap among the convex hulls increases. Figure 2.9 shows two sets of curve segments, one of which has very little, and the other a lot of geometric coherence.

It may sound awkward that a model in which there is more space in-



(a) Little coherence (b) Much coherence

Figure 2.9: The amount of geometric coherence in a set of curve segments

between the components, has more geometric coherence than a model in which all objects are interlinked. We should keep in mind that geometric coherence has to do with the degree in which a location in space can be associated with one designated object, rather than the degree in which objects cohere.

A **bounding volume** is a simple primitive that encloses a more complex shape, and for which a cheap intersection test exists. If the probability that the bounding volumes of the objects intersect is low, i.e., there is a lot of geometric coherence among the objects in a model, we may save ourselves some time by first testing the bounding volumes for intersection. Only for the objects whose bounding volumes intersect, we need to perform an exact and expensive intersection test. Again, the use of bounding volumes requires some additional storage and computational cost, so performance is gained only if the bounding volumes have a high probability of being disjoint.

2.4.3 Average Time

In order to attain the best average performance by exploiting coherence, collision detection is typically done as a sequence of intersection tests of increasing computational cost. Except for the last test, each test establishes an answer to the intersection query only for a portion of all possible configurations of objects. The last test must return an answer for all the remaining configurations. Here, an answer may be either positive or negative. A test in the sequence needs to be performed only if the previous test failed, i.e., it did not establish an answer to the query.

For instance, an intersection test of two geometric objects may consist of a bounding volume test and an exact intersection test. If the bounding volumes of the two objects do not intersect then a negative answer is returned, thus, the bounding volume test is successful. Otherwise, we need to do an exact intersection test.

Let $S_1, \ldots, S_n$ be a sequence of intersection tests that is used for answering an intersection query, and let f_i represent the event that test S_i fails, and C_i, the average time necessary for performing S_i. We can define the average time of an intersection query as

$$T_{\text{avg}} = \sum_{i=1}^{n} P[f_1 \cdots f_{i-1}]C_i$$

where $P[f_1 \cdots f_{i-1}]$ is the probability of failure of tests $S_1, \ldots, S_{i-1}$ for a given input domain. Often, C_i depends on the size of the input. For instance, testing whether two polygons intersect takes an amount of time that is linear in the total number of vertices of the two polygons (cf. Chapter 3). Hence, the average time for performing an intersection query is often a function of the input size.

It is our goal to design intersection queries as a sequence of tests for which the average time is minimized for a realistic input domain. Since we will be performing a lot of these intersection queries per frame, we will attain the best performance by keeping the average query cost low. It can be seen that the best times are gained by using tests S_i for which both $P[f_i | f_1 \cdots f_{i-1}]$, i.e., the probability of the test failing under the condition that the former tests failed, and C_i are small.

The next question to answer is how the values for the probabilities and cost are determined. The cost values may be found by counting the number of primitive operations in a single intersection test. Primitive operations are for instance arithmetic operations (additions, subtractions, multiplications, and divisions), branch instructions, and memory accesses. We often express the cost of a test routine by the number of arithmetic operations only.

For more complex intersection tests that depend on the input size, expressing the cost in primitive operations is often not feasible. In these cases, we determine the cost empirically. For a given implementation and input size, the cost is measured by running a large number of tests for a representative input domain using a profiling tool, such as *gprof* [32]. Of course, the measured values are only valid for the given implementation, input size, and testing platform, and thus, we need to apply our findings with some restrictions.

The probability values are also best found empirically by running a large number of tests. Although a field of probability theory called *stochastic geometry* [90] can be applied for determining these values for some configurations of objects, the majority of object types and input domains are too complex in order for an analytic approach to be feasible. The probability values can be found simply by counting the number of times each intersection test is called. For this purpose, we may also use a profiling tool.

Chapter 3

Polygons

"Elementary, my dear Watson."
Sherlock Holmes

Polygons are currently the most commonly used modeling primitives in 3D computer animation. Hence, finding collision detection methods for polygonal objects is our first concern. In this chapter, we discuss a number of intersection tests on non-convex polygons. We will present new algorithms for detecting and computing an intersection between a pair of polygons, after which we discuss algorithms for testing the intersection of a polygon with some other primitives. Let us first examine the problem of detecting intersections of general non-convex polyhedra in greater detail.

3.1 Colliding Polyhedra

A polyhedron is an object that has a polyhedral surface as boundary. For the purpose of visualization, a polyhedron is usually represented by a collection of polygons. In Chapter 4, we will discuss algorithms for collision detection of convex polyhedra. Here, we will look into the problem of detecting collisions between non-convex polyhedra.

3.1.1 Convex Decomposition

One way to tackle the problem is by decomposing non-convex polyhedra into convex subparts, and use an algorithm for convex polyhedra on the subparts. However, convex decomposition of general polyhedra is more difficult than its two-dimensional counterpart, convex decomposition of

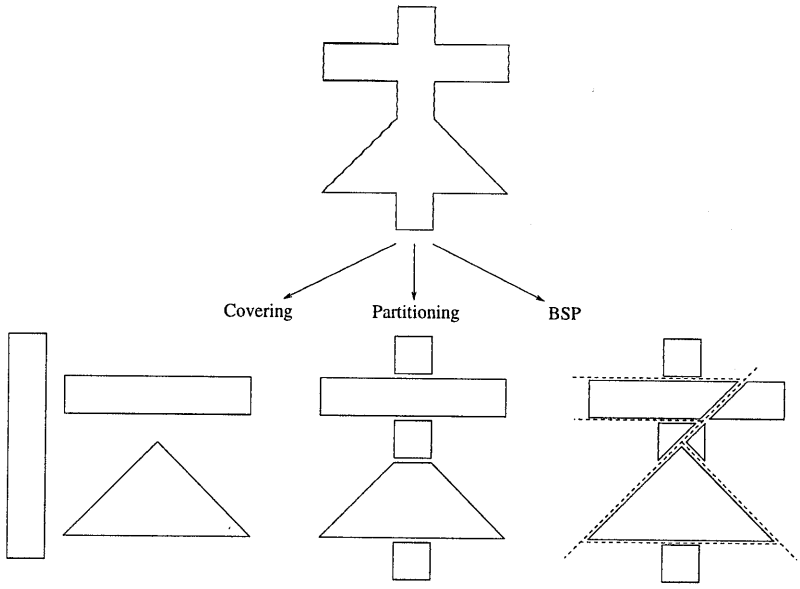


Figure 3.1: Convex decomposition methods

polygons. In particular, for some polyhedra it is impossible to be tetrahedralized, i.e., partitioned into tetrahedra, such that each tetrahedron's vertices are vertices of the polyhedron. In fact, polyhedra exist which require $\Omega(n^2)$ convex pieces in the best partitioning, where n is the number of vertices [76].

An easy implementable partitioning method is **binary space partitioning** (BSP) [95]. A BSP is constructed by recursively subdividing a space into convex regions, called cells, using well-chosen planes. By choosing the partitioning planes from the set of supporting planes of the polyhedron's facets, a BSP can be formed such that the polyhedron is the union of a subset of the BSP's cells. A drawback of the BSP method is that it often results in unnecessarily many components. We will further discuss the use of BSPs for representing polyhedra in Chapter 5.

It is worth noting that, for the purpose of collision detection, it is not necessary to partition the polyhedra. We may decompose a polyhedron into overlapping pieces, as depicted in Figure 3.1. The question remains unanswered whether this added freedom yields decompositions that have significantly fewer components. Surprisingly little is found in literature on this intriguing problem.

Convex decomposition is still an important research topic. Existing partitioning methods often generate an unacceptable number of convex

pieces ($O(n^2)$ for a polyhedron having n vertices), in order for a convex component intersection approach to be useful.

3.1.2 Boundary Intersections

A more feasible approach is to focus on boundary intersections. Configurations of colliding polyhedra can be classified in the following two categories:

1. The boundaries of the polyhedra intersect, i.e., there exists a pair of intersecting polygons, one from each polyhedron.

2. One polyhedron is contained in the other polyhedron's interior. In this case the boundaries do not intersect.

A basic strategy for finding an intersection between polyhedra is the following. First, test if a pair of polygons, one from each polyhedron, intersect. If such a pair exists then the polyhedra intersect. Otherwise, test for each polyhedron whether it contains a point from the other polyhedron. It does not matter which point of a polyhedron is tested, since under the condition that the boundaries do not intersect, it is necessary that if a point of a polyhedron is contained in the other, then this polyhedron is completely enclosed by the other polyhedron.

A point-in-polyhedron test can be done similar to its 2D variant, a point-in-polygon test. A ray with the query point as its starting point extending infinitely in one direction crosses the boundary an even number of times if the query point lies outside and an odd number of times if the query point lies inside the polyhedron. Hence, by counting the number of times the ray crosses the boundary we can classify the location of a given a query point with respect to a polyhedron.

Although rather straightforward in theory, this method is quite tricky in practice. Due to precision problems, degenerate cases, where the ray passes (nearly) through an edge of the boundary, may cause incorrect results. O' Rourke presents a robust implementation of the ray intersection count in [77]. He uses an approach in which a randomly generated ray is used. If the chosen ray happens to result in a degeneracy, then another random ray is tried. This process is repeated until a ray is found that does not result in a degeneracy.

In computer animation, a point containment test is often not necessary. If objects move with a limited velocity, and if they have a large enough size, it will be impossible for a pair of objects to travel in one frame from a disjoint configuration to a configuration in which one object is enclosed by

the other. In these cases, a boundary intersection must occur before one of the objects is enclosed by the other. By immediately resolving any collision occurring, a pair of objects will not get into a containment configuration. Hence, it is unnecessary to test for such configurations.

3.2 Polygon-Polygon Intersections

As we saw, testing for the intersection of a pair of polyhedral surfaces involves testing pairs of polygons for intersection. In order to reduce the number of polygon-polygon intersection tests we can apply a spatial data structure. In Chapter 5, we will discuss a number of data structures that may be used for this purpose. For now, let us focus on the polygon-polygon intersection test.

In this section we discuss two algorithm for testing the intersection of a pair of non-convex polygons in three-dimensional space. Both algorithms use a plane equation of the polygons' supporting planes. When transforming a polygon to another coordinate system, it is usually faster to transform its plane equation, rather than recompute it for the new coordinates. A plane equation can be transformed using the method described in Appendix A.

3.2.1 A Straightforward Approach

A common way of detecting an intersection of two polygons is testing each edge of the first polygon against the second polygon for intersection and vice versa [10]. Clearly, this suffices to find an intersection, since for a pair of intersecting polygons at least one polygon has an edge that intersects the other. An edge-polygon intersection test involves the following steps. First, the point of intersection of the edge with the polygon's supporting plane is computed, after which this point is tested for containment in the polygon.

Finding the point of intersection of an edge and a plane involves computing the signed distances of the edge's endpoints. For a plane

$$H(\mathbf{n}, \delta) = \{\mathbf{x} \in \mathbb{R}^3 : \mathbf{n} \cdot \mathbf{x} + \delta = 0\},$$

where $\|\mathbf{n}\| = 1$, the signed distance of a point $\mathbf{p}$ to this plane is $\mathbf{n} \cdot \mathbf{p} + \delta$. Note that for the correctness of the following computations, it is not necessary that $\|\mathbf{n}\| = 1$. The length of $\mathbf{n}$ needs to be one only if we want the absolute value of the signed distance to be equal to the actual Euclidean distance to the plane.

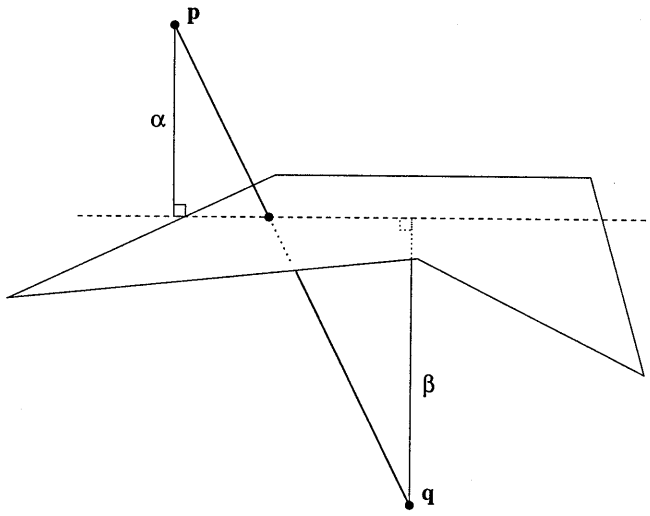


Figure 3.2: Computing the point of intersection of an edge and a polygon's supporting plane

Let α and β be the signed distances of the endpoints of the edge. If α and β have the same sign, i.e. the endpoints lie on the same side of the plane, then the edge does not intersect the polygon and can be rejected. If α and β have opposite signs, then the intersection point of the edge and the plane is the point $\mathbf{p}+\lambda(\mathbf{q}-\mathbf{p})$, where $\lambda = \alpha/(\alpha-\beta)$. Figure 3.2 illustrates this operation.

The next step after computing the point of intersection of the edge and the supporting plane of the polygon, is testing whether this point is contained in the polygon. Since this is a 2D problem, we need to project all points onto a plane, i.e., we need to drop (ignore) a coordinate in the 3D coordinates of the intersection point and the vertices of the polygon. The safest coordinate axis to drop is the one whose angle with the normal of the polygon's supporting plane is the smallest. Since the projection of the polygon onto the plane orthogonal to this axis has the largest area of all coordinate axes, we avoid the problem of the projection of the polygon being a line segment (area is zero) [41]. The coordinate axis whose angle with the normal of the polygon's supporting plane is the smallest corresponds with the coordinate of the plane's normal that has the largest absolute value of the three coordinate values. The axis is referred to as the **closest axis** to the plane normal.

A discussion of a number of algorithms for point-in-polygon tests is presented by Haines in [50]. When preprocessing of the polygons is not

allowed, the best choice of a point-in-polygon test for non-convex polygons is the crossings test. The crossings test counts the number of times a ray, that originates from the query point, crosses boundary of the polygon. If the number of crossings is odd then the query point lies inside the polygon, otherwise, it lies outside the polygon. Haines includes a fast and robust implementation of this algorithm in the article. This point-in-polygon test takes linear time in the number of vertices.

Using this approach, a polygon-polygon intersection test takes $O(n^2)$ time in the worst case for a pair of polygons with n vertices each, since each edge of each polygon may intersect the other polygon's supporting plane. This bound may be reduced to $O(n \log n)$, by first computing all intersection points of one polygon's edges with the other polygon's supporting plane, and then performing all point-in-polygon tests at once, by applying a plane-sweep algorithm. Yet, in the following text, we present a new approach for polygon-polygon intersection testing, which yields a worst-case $O(n)$ time algorithm.

3.2.2 A New Approach

Our approach is based on the following idea. A pair of polygons intersect iff the intersections of each polygon and the other polygon's supporting plane overlap. The intersection of a non-convex polygon and a plane is a collection of collinear line segments. We simply ignore intersections of coplanar polygons, i.e., we assume an infinitesimal distance between the parallel polygons. Ignoring this case is not harmful for detecting collisions between the boundaries of a pair of polyhedra, since if two coplanar polygons from a pair of polyhedra intersect, then other intersecting pairs of polygons must exist that are not coplanar.

The line segments for both polygons coincide with the line of intersection of the polygons' supporting planes, as depicted in Figure 3.3. We see that for a pair of intersecting polygons, one or more pairs of line segments overlap.

The intersection of a non-convex polygon and a plane is computed using a polygon clipping technique [92, 42]. Using the edge-plane intersection we saw earlier, we find all intersection points of polygon edges with the plane. These intersection points are the endpoints of the line segments. The endpoints are computed in the order in which they appear along the boundary, as can be seen in Figure 3.4. In order to find the intersection line segments we need to sort the endpoints along the line of intersection.

The direction of the line of intersection can be computed by taking the

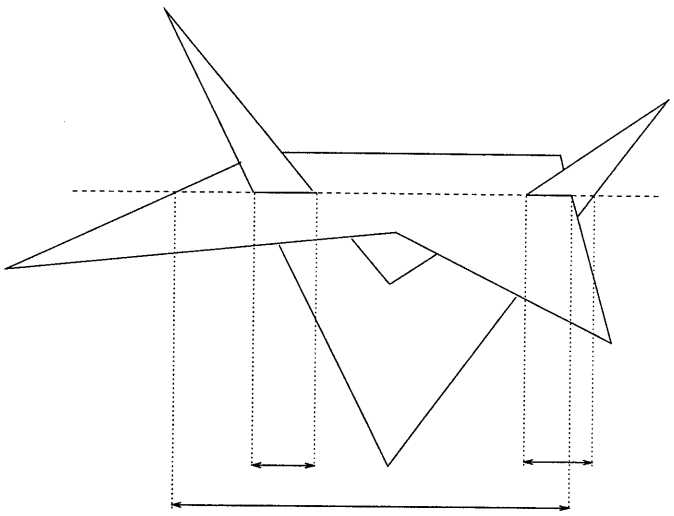


Figure 3.3: The intersection of a pair of polygons

cross product of the normals of the polygons' supporting planes. A sequence of points is sorted along this line in the following way. First, we determine the closest axis to the direction of the line of intersection. It is easier to 'project' a three-dimensional point onto a coordinate axis than onto an arbitrary axis, since for a coordinate axis the projection is simply the corresponding coordinate of the point. We choose the closest axis to the direction of the line of intersection, in order to avoid the case where the coordinate axis is perpendicular to the line of intersection. Sorting the endpoints by their coordinate on an axis that is perpendicular to the line of intersection will give an incorrect result, since all endpoints have the same coordinate for this axis.

Next, the endpoints are sorted according to their coordinate values on this axis. For this purpose, we can apply a general sorting algorithm, such as Quicksort, which has $O(n \log n)$ average time complexity for a sequence of n points. However, for this particular sorting problem, which is referred to as **Jordan sorting**, we can find an algorithm with a better time-complexity. Jordan sorting is the problem of sorting a sequence of intersection points of a Jordan curve with an axis, given in the order in which they occur along the curve. Clearly, this is the case here, since the boundary of a simple polygon is a Jordan curve. It has been shown that Jordan sorting has only a linear time complexity [53, 36]. However, these algorithms require rather sophisticated data structures. Therefore, we regard it unlikely that Jordan sorting outperforms the Quicksort routine for

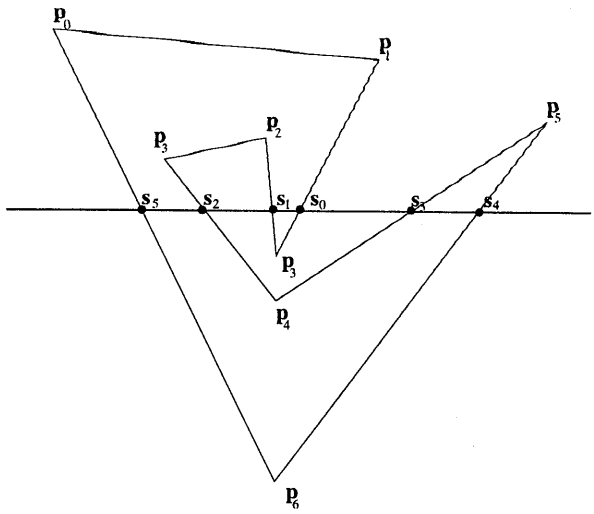


Figure 3.4: The intersection of a non-convex polygon and a plane

the input sizes that we are interested in.

After sorting the endpoints, we have a representation of the intersection line segments in the order in which they occur along the closest axis. Let $\text{conv}\{\mathbf{s}_{2i}, \mathbf{s}_{2i+1}\}$ and $\text{conv}\{\mathbf{t}_{2j}, \mathbf{t}_{2j+1}\}$ be the sorted sequences of intersection line segments of the two polygons, and let $[\sigma_{2i}, \sigma_{2i+1}]$ and $[\tau_{2j}, \tau_{2j+1}]$ be their respective projections the closest axis, where $0 \leq i < l$ and $0 \leq j < m$. Overlap among the segments is found by simultaneously scanning the sorted sequences of endpoints for both polygons. Two segments overlap iff their projections onto the closest axis overlap. More precisely, the i-th segment of $\mathbf{s}$ overlaps the j-th segment of $\mathbf{t}$ iff $[\sigma_{2i}, \sigma_{2i+1}] \cap [\tau_{2j}, \tau_{2j+1}] \neq \emptyset$, i.e., $\sigma_{2i} \leq \tau_{2j+1}$ and $\tau_{2j} \leq \sigma_{2i+1}$. Furthermore, assume without loss of generality that $\sigma_{2i+1} < \tau_{2j}$. Then, the i-th segment of $\mathbf{s}$ does not overlap the k-th segment of $\mathbf{t}$, where $k \geq j$. This property is exploited in Algorithm 3.1 for detecting whether two sequences of line segments overlap in $O(l + m)$ time. For a pair of polygons, each having n vertices, we have $l \leq n$ and $m \leq n$, since each edge contributes at most one endpoint. Hence, Algorithm 3.1 runs in $O(n)$ time.

Let us summarize the complete algorithm for testing the intersection of a pair of non-convex polygons, each having n vertices.

1. First, the direction of the line of intersection of the polygon's supporting planes is computed. If the planes are parallel, then false is returned. We simply ignore the degenerate case where the planes

Algorithm 3.1 Detecting overlap in two sequences of line segments

```
i := 0;
j := 0;
while i < l and j < m do begin
  if σ_{2i+1} < τ_{2j} then i := i + 1
  else if τ_{2j+1} < σ_{2i} then j := j + 1
  else return true
end;
return false
```

coincide, assuming an infinitesimal distance between the planes.

2. Next, for each polygon, the intersection points of its edges with the other polygon's supporting plane are computed and stored in a list. This takes $O(n)$ time.

3. The lists of intersection points are sorted along the coordinate axis that is closest to the direction of the line of intersection. This takes $O(n)$ time using Jordan sorting.

4. Finally, the lists are scanned using Algorithm 3.1 in order to detect a pair of overlapping segments. This operation also takes $O(n)$ time.

We see that the complete algorithm has a worst-case time-complexity of $O(n)$. At this level though, asymptotic worst-case performance bounds are mainly of theoretical interest.

In practice, we will in most cases perform a bounding box test before testing the polygons for intersection, in which case the performance of the two approaches differs only slightly. In our experiments, we found that the second approach using Quicksort is only 10% faster than the first approach in the cases where the bounding boxes of the polygons intersect. The better performance is due to the fact that with the second approach, an early exit is possible if one of the polygons does not intersect the other polygon's supporting plane, whereas the first approach may still perform a number of point-in-polygon tests.

As a conclusion, we present an algorithm for computing the intersection of a pair of non-convex polygons, which is a modification of the detection algorithm. The main difference with intersection detection is that for intersection computation we need to have all intersecting pairs of line

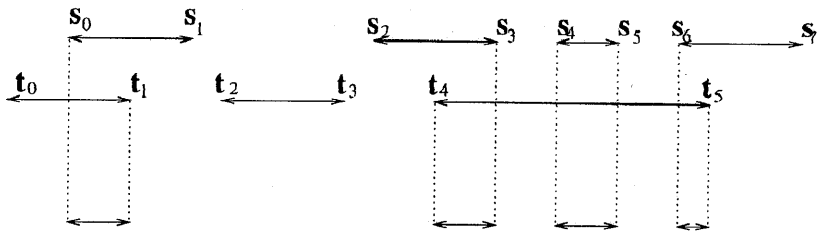


Figure 3.5: Intersection of two line segment sequences

segments. Note that a single segment from one polygon may overlap multiple segments of the other polygon, as can be seen in Figure 3.5. As soon as a pair of overlapping segments is found the segment of intersection of these segments is reported. After reporting an overlapping pair of segments, we examine the next pair of segments by progressing in one of the segment sequences. Assume without loss of generality that $\sigma_{2i+1} < \tau_{2j+1}$. Since the sequences are sorted we know that the i-th segment in $\mathbf{s}$ does not overlap any of the k-th segments of $\mathbf{t}$ for $k > j$. Hence, we discard the i-th segment of $\mathbf{s}$ from consideration, and progress to the next segment in $\mathbf{s}$. See Algorithm 3.2 for a description of the intersection computation algorithm in pseudo-code. As the detection algorithm, the intersection computation algorithm runs also in $O(l+m)$ time, which is $O(n)$ for a pair of polygons of n vertices each.

Algorithm 3.2 Computing the intersection of two sequences of line segments

$i := 0;$
$j := 0;$
while $i < l$ **and** $j < m$ **do begin**
  if $\sigma_{2i+1} < \tau_{2j}$ **then** $i := i + 1$
  else if $\tau_{2j+1} < \sigma_{2i}$ **then** $j := j + 1$
  else begin
    $\mathbf{b} :=$ **if** $\sigma_{2i} > \tau_{2j}$ **then** $\mathbf{s}_{2i}$ **else** $\mathbf{t}_{2j}$;
    $\mathbf{e} :=$ **if** $\sigma_{2i+1} < \tau_{2j+1}$ **then** $\mathbf{s}_{2i+1}$ **else** $\mathbf{t}_{2j+1}$;
    "report the line segment connecting **b** and **e**";
    if $\sigma_{2i+1} < \tau_{2j+1}$ **then** $i := i + 1$ **else** $j := j + 1$
  end
end

We briefly discuss some alternative algorithms for polygon-polygon intersection testing. The algorithms we have seen so far use a representation of the supporting plane. Since a plane equation is represented using only four scalars, the additional storage that is needed for representing the plane equation is in most cases acceptable. However, transformations on polygons, which are needed for getting the coordinates of the polygons relative to the coordinate system of reference, require some additional overhead if plane equation are involved. See Appendix A for details on how to compute the image of a plane equation under affine transformations.

Hence, using an intersection testing algorithm that does not require a representation of the plane equation might be a valid option in some applications. In [96] Thomas and Torras present a variant of our first approach that does not require a plane representation. In their algorithm, the point of intersection of an edge and a plane is not computed. Instead, an edge-polygon test is performed using only dot and cross products, and sign comparisons. Held proposed a similar approach in [52] for testing the intersection of an edge and a triangle.

The second approach that we saw can be simplified for convex polygons. Since the intersection of a convex polygon and a plane is a single line segment, sorting of intersection points is reduced to a single comparison and swap. Möller exploits this property in [67] in which he presents an algorithm for testing the intersection of triangles, although his algorithm is readily applicable to convex polygons in general. We will see other intersection detection algorithms for convex polygons in Chapter 4.

3.3 Polygon-Volume Intersections

In some cases we need to test the intersection of a polygon and a convex primitive volume. For instance, in virtual environment simulations, such as performed by VRML browsers [8], the avatar is usually represented by a volume, such as a box, a cylinder, or a sphere, and the scene is usually represented by polyhedra and polygonal surfaces. Here, collision handling is necessary in order to prevent the avatar from walking through obstacles, such as walls. Another example is view frustum culling, which is used to improve the rendering performance of a 3D graphics engine by processing only those polygons in the scene that intersect with the camera's field of view. Closer to our problem domain, polygon-volume intersection tests are useful in intersection testing between complex polygonal models for culling the polygons of one model that do not intersect with a bounding

volume of the other model.

The basic strategy for polygon-volume intersection testing is described as follows:

1. If the polygon's supporting plane and the volume do not intersect, then return **false**. Otherwise,

2. if a vertex of the polygon is contained in the volume, then return **true**. Otherwise,

3. if an edge of the polygon intersects the volume, then return **true**. Otherwise,

4. if a point common to both the volume and the polygon's supporting plane is contained in the polygon, then return **true**. Otherwise, return **false**.

This strategy can be used for testing the intersection of a polygon with any object type, and is most useful for intersection tests of a polygon with a primitive volume, such as a sphere or a box, since for these types of objects the individual tests in the strategy are quite elementary. For polygon-polygon intersection testing we found strategies that exploit the symmetry of this problem to be more effective. Note that the first two tests in this algorithm are not necessary for the correctness of the operation, however they contribute to the performance of the test to a large extent. Recall that it is our aim to come up with algorithms that have good average performance. In this light, the two tests are quite useful, since they give us a quick answer for the majority of intersection tests and are relatively cheap.

We will show how to apply this basic strategy to polygon-sphere and polygon-box intersection testing. For other primitives, such as cylinders and cones, the algorithms are similar to the polygon-sphere intersection test. We discuss the polygon-box intersection test separately, since it allows further optimizations. Polygon-box intersection algorithms that follow this strategy are found in [98, 47].

3.3.1 Polygon-Sphere Intersection Testing

Deciding whether a sphere intersects a plane is quite simple. The sphere intersects the plane if the distance of its center to the plane is at most its radius. Let $\mathbf{c}$ be the center of the sphere and $H(\mathbf{n}, \delta)$ the polygon's supporting plane. Then, the distance of $\mathbf{c}$ to the plane is $|\mathbf{n} \cdot \mathbf{c} + \delta| / \|\mathbf{n}\|$. The point containment and segment intersection test are performed similarly

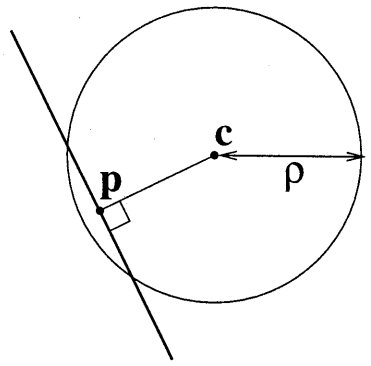


Figure 3.6: Finding a point **p** common to a sphere and a plane

by computing the distance from the vertex or edge to the center of the sphere.

For the final test a common point to both the plane and the sphere needs to be found. As common point we take the center of the sphere projected onto the plane, i.e., the intersection point of the plane and the line orthogonal to the plane passing through the center. This point is denoted by **p** in Figure 3.6. It can be seen that under the assumption that the plane intersects the sphere, this point is indeed contained in both the plane and the sphere. Finally, the intersection point is tested for containment in the polygon, for which we use the point-in-polygon test we saw in the context of edge-polygon intersection testing.

3.3.2 Polygon-Box Intersection Testing

For polygon-box intersection testing we use the same basic strategy, and apply a number of optimizations in order to further improve performance. We assume the box is aligned to the coordinate system of reference. Let **c** be the center, and $\mathbf{e} = (\eta_1, \eta_2, \eta_3)^T$, the extent vector of the box, and let $H(\mathbf{n}, \delta)$ be the polygon's supporting plane, and assume $\|\mathbf{n}\| = 1$. We test whether the plane intersects the box as follows. The vertices of the box are the points $\mathbf{c} + (\pm\eta_1, \pm\eta_2, \pm\eta_3)^T$. Hence, the projection of the box onto **n** is the interval $[\mathbf{n} \cdot \mathbf{c} - \rho, \mathbf{n} \cdot \mathbf{c} + \rho]$, where $\rho = \max\{\mathbf{n} \cdot (\pm\eta_1, \pm\eta_2, \pm\eta_3)^T\}$. We see that for $\mathbf{n} = (\nu_1, \nu_2, \nu_3)^T$, the value of ρ is $|\nu_1\eta_1| + |\nu_2\eta_2| + |\nu_3\eta_3|$. See Figure 3.7 for a visualization of this projection. It can be seen that the plane intersects the box iff the distance of the center to the plane, which is $|\mathbf{n} \cdot \mathbf{c} + \delta|$, is at most ρ.

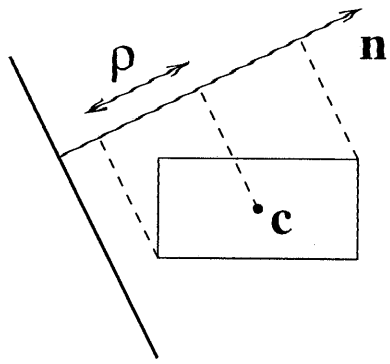


Figure 3.7: Testing the intersection of a box and a plane

For point containment and edge intersection testing we enter the realm of line segment clipping for which a considerable amount of literature is available (cf. [60, 33, 9, 29, 30]). In contrast with line clipping, where the intersection needs to be computed, we only need to detect an intersection. Nevertheless, the algorithm we propose has a lot of similarities with common line clipping algorithms. We borrow techniques from two popular line clippers: Cohen-Sutherland (CS) and Liang-Barsky (LB) [60, 33]. Originally, CS and LB were presented for clipping in 2D, however both algorithms can be readily generalized to 3D. We will give brief description of our algorithm.

CS uses a classification of points according to the six planes supporting the facets of the box, each plane is oriented such that its normal is pointing outward. The classification is represented by a six bit code, in which each bit corresponds to a plane. The code is referred to as **outcode**. A bit in the outcode is 1 if the point lies in the positive open halfspace of the corresponding plane, and 0 otherwise. We see that any point that is contained in the box is classified as 000000. Thus, the point containment test in our basic polygon-volume strategy is performed simply by testing whether the outcode of a vertex is zero. Furthermore, if the outcodes of the edge's endpoints contain the same bit, i.e., their bit-wise 'and' is nonzero, the edge can be rejected, since the corresponding plane separates the edge from the box. Assume that neither endpoint is contained in the box, and their outcodes 'and' to zero. Then, we need to test whether the edge intersects the box, as illustrated in Figure 3.8 in the 2D case.

For the edge-box test, the third test in our basic strategy is performed using a technique from the LB parametric line clipping algorithm. In LB

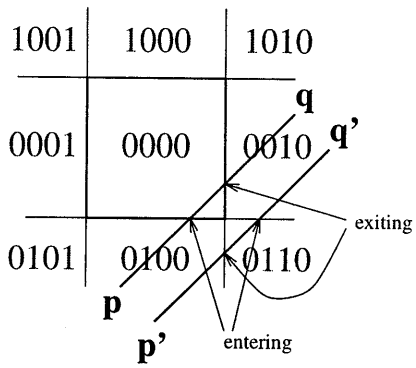


Figure 3.8: Edge-box intersections in Cohen-Sutherland line clipping

the parameters of the intersection points of the edge with the box planes are computed. Let $\mathbf{p}$ and $\mathbf{q}$ be the endpoints of the edge. Then, an intersection point can be expressed as $\mathbf{p} + \lambda(\mathbf{q} - \mathbf{p})$, where λ is its parameter. By classifying the intersection point as 'entering' or 'exiting' we can decide whether the edge penetrates the box. An intersection point is classified as 'entering' if moving from $\mathbf{p}$ to $\mathbf{q}$ we go from the positive to the negative halfspace of the corresponding plane, and 'exiting' otherwise. It can be seen that an edge intersects the box iff the largest parameter of an 'entering' intersection point is at most the smallest parameter of an 'exiting' intersection point.

We will now explain why CS and LB make such a good team. First of all, under the assumption that the outcodes of the endpoints 'and' to zero, the '1' bits in both outcodes correspond to those planes for which intersection points need to be computed, since only for these bits the endpoints lie in opposite halfspaces of the corresponding plane. Moreover, each intersection point corresponding to a bit in the outcode of $\mathbf{p}$ is 'entering', and each intersection point corresponding to a bit in $\mathbf{q}$'s outcode is 'exiting'. Hence, the edge intersects the box iff the largest parameter corresponding to a bit in $\mathbf{p}$'s outcode is at most the smallest parameter corresponding to a bit in $\mathbf{q}$'s outcode. We see how neatly LB benefits from the outcodes computed by CS. Considering the long history of line clipping, it is remarkable that a hybrid of CS and LB got published only recently [9]).

We have yet another use for the outcodes in our polygon-box intersection detection algorithm. In order to get a fast answer for our intersection query, it is useful to test if all the polygons vertices lie in the positive halfspace of any of the six planes. For this purpose, we bit-wise 'and' the out-

codes of all the vertices, and test whether the result is nonzero, in which case we return **false**, and exit. The test is best done before the edge-box tests, since it is fairly cheap and results in a rejection in a large number of cases.

Finally, if none of these tests results in an answer, there is still a possibility that the box intersects the interior of the polygon. This case is tested by the fourth test in the basic strategy. In order to test this case, we compute the intersection point of the polygon's plane and the box diagonal whose angle with the plane is largest, and check whether this point lies inside the polygon.

The latter test is proposed by Green and Hatch in [47] as an improvement over the test in the approach presented by Voorhies in [98]. The approach presented in [47] differs from ours with respect to the edge-box intersection test. In the approach by Green and Hatch an edge-box intersection is detected by testing whether the origin is contained in the Minkowski sum of the edge and the box. The Minkowski sum of a line segment and a box is a rhombic dodecahedron, which is a 6-DOP, according to the definition in Chapter 2. The origin is tested for containment in the Minkowski sum by testing whether the origin lies in-between parallel planes for each of the six plane orientations. This test is likely to be faster and more robust than our CS-LB clipping test, since for the Green-Hatch approach there are no divisions necessary, whereas in the clipping approach divisions are necessary for computing the line parameters. However, the Green-Hatch approach does not allow common point computation in a straightforward manner. Therefore, our CS-LB approach is the best choice if a witness to an intersection of a polygon and a box is required. We will see more on Minkowski-sum-based algorithms in Chapter 4.

Chapter 4

Convex Objects

"Just ask the axis."
Jimi Hendrix

In this chapter we study a number of algorithms that exploit convexity. The algorithms are characterized according to their witness types. We will discuss intersection detection algorithms for finding either a common point or a separating axis, and distance computation algorithms. Most of the discussed distance algorithms can be tailored to return a closest point pair. We conclude with a discussion of the Gilbert-Johnson-Keerthi distance algorithm (GJK), for which we present an implementation that has improved performance, robustness, and versatility over earlier implementations. We will show how GJK can be tailored to efficiently find a common point, a separating axis, and a pair of closest points for general convex objects. Except for GJK, all algorithms discussed here are applicable to polytopes only.

4.1 Finding a Common Point

We saw that the basic strategy for detecting intersections between polygons was searching for a common point. Let us examine how far this strategy gets us when applied to convex polyhedra.

The first known algorithm that improved upon the trivial $O(n^2)$ bound was presented by Muller and Preparata [69]. Their algorithm requires that the polyhedra are represented by a doubly-connected edge list (DCEL), which represents the adjacency graph of the vertices of a polyhedron in a convenient way. The detection algorithm determines a common point

in $O(n \log n)$ time, where n is the total number of vertices of both polyhedra. This algorithm is used for constructing the intersection of two convex polyhedra in $O(n \log n)$ time.

In fact, it has been shown by Chazelle and Dobkin that detecting intersections between polyhedra can be done in sub-linear time if the proper preprocessing on the polyhedra is allowed, whereas constructing the intersection of polyhedra has a linear lower bound [17]. The best upper bound for the common point detection problem was given by Dobkin and Kirkpatrick [26]. Their algorithm has an upper bound of $O(\log^2 n)$, and requires a representation of each polyhedron decomposed into drums, which are the convex subparts that are formed by slicing the polyhedron at each vertex by a horizontal plane. The drum representation requires $O(n^2)$ space. Little is known of how well these algorithms perform on current computer platforms, since implementations are not available and probably nonexistent.

An interesting way to view the intersection detection problem for polytopes is by regarding it as a linear programming (LP) problem. An LP problem is an optimization problem of the form

$$\begin{aligned} &\text{maximize} && \mathbf{c} \cdot \mathbf{x} \\ &\text{subject to} && \mathbf{v}_i \cdot \mathbf{x} + \delta_i \leq 0 \quad \text{for } i = 1, \ldots, n, \end{aligned}$$

where the vector $\mathbf{c}$, and the constraints $\mathbf{v}_i \cdot \mathbf{x} + \delta_i \leq 0$ are given, and $\mathbf{x}$ is the variable for which an optimal solution is sought. The feasible set is the intersection of all halfspaces $H^-(\mathbf{v}_i, \delta_i)$. A feasibility test returns, if possible, a member of the feasible set. For a feasibility test, the objective vector $\mathbf{c}$ may be taken to be any vector, including the zero vector.

The problem of finding a common point of a pair of polytopes can be expressed as an LP feasibility test in the following way. We take the halfspaces of the two polytopes as the constraints of our LP problem, thus, the intersection of the polytopes is the feasible set. Recall that the number of halfspaces in a halfspace representation of a polytope is linear in its number of vertices. Hence, for a pair of polytopes of n vertices each, we have $O(n)$ constraints. Clearly, a common point of a pair of polytopes is returned by a feasibility test on the set of halfspaces of both polyhedra.

Both Meggido [63] and Dyer [31] showed independently that low dimension LP problems can be solved in linear time with respect to the number of constraints. Their solutions however, are rather complex and have a large constant for problems in three or higher dimensions. Low-dimensional LPs can be implemented in a surprisingly simple manner, by applying a randomized algorithm, as shown by Seidel in [84]. His algorithm has expected linear time complexity.

	MP [69]	DK [26]	LP problem
Representation	DCEL	drum decomp.	halfspaces
Space bound	$O(n)$	$O(n^2)$	$O(n)$
Time bound	$O(n \log n)$	$O(\log^2 n)$	expected $O(n)$
Implementation	unknown	unknown	Seidel [84]

Table 4.1: Common point search algorithms

Table 4.1 shows an overview of the discussed common point detection algorithms. These algorithms seem less useful for collision detection of convex polyhedra, since we suspect that they have a large constant for their time bounds. Moreover, exploiting coherence by using common points from previous frames for speeding up intersection tests does not appear to be useful, since in most applications of collision detection, collisions are resolved rather than maintained.

4.2 Finding a Separating Axis

Another strategy for detecting collisions between convex objects is searching for a separating plane or a separating axis. A separating plane or axis is a witness of the disjointness of a pair of objects. This strategy is often better suited for exploiting coherence than common point search, since it is the objective in most applications to keep object pairs disjoint. Hence, a witness of the disjointness of a pair of objects is likely to persist over several frames.

A **separating plane** of two objects is a plane for which one object lies in the positive, and the other in the negative open halfspace. The axis orthogonal to a separating plane is referred to as a **separating axis**. Let the plane $H(\mathbf{v}, \delta)$ be a separating plane of objects A and B, and assume without loss of generality that $A \subset H^{\oplus}(\mathbf{v}, \delta)$ and $B \subset H^{\ominus}(\mathbf{v}, \delta)$. We see that for separating axis $\mathbf{v}$ we have

$$\mathbf{v} \cdot \mathbf{x} > \mathbf{v} \cdot \mathbf{y} \quad \text{for all } \mathbf{x} \in A \text{ and } \mathbf{y} \in B.$$

Conversely, for an axis $\mathbf{v}$ for which the above inequality holds, we find that each plane $H(\mathbf{v}, \delta)$ with $\max\{\mathbf{v} \cdot \mathbf{y} : \mathbf{y} \in B\} < \delta < \min\{\mathbf{v} \cdot \mathbf{x} : \mathbf{x} \in A\}$ is a separating plane.

Coherence can be exploited by testing whether a separating plane or axis from a previous frame also separates the objects in the current frame. This operation is usually a lot cheaper than recomputing a separating

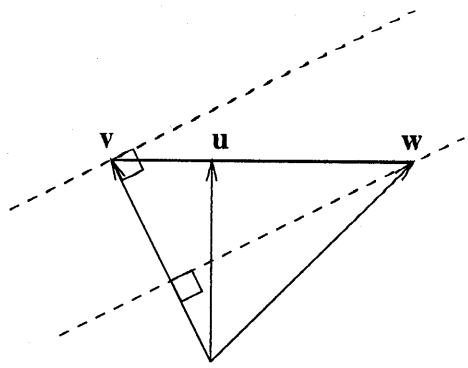


Figure 4.1: If $\mathbf{v} \cdot \mathbf{w} - \|\mathbf{v}\|^2 < 0$, then conv{$\mathbf{v}$, $\mathbf{w}$} contains a point $\mathbf{u}$ for which $\|\mathbf{u}\| < \|\mathbf{v}\|$.

plane or axis. Since animated objects usually move relatively little in-between frames, a separating axis from a previous frame is likely to be a separating axis in the current frame, in which case we save ourselves the cost of recomputing a separating axis.

We will show that for nonintersecting convex objects a separating axis always exists. We do this by showing that for a pair of closest points of $\mathbf{a} \in A$ and $\mathbf{b} \in B$ of nonintersecting convex objects A and B, the vector $\mathbf{a} - \mathbf{b}$ is a separating axis.

Lemma 4.1. *Let* $\mathbf{v}$ *and* $\mathbf{w}$ *be points, and suppose that* $\mathbf{v} \cdot \mathbf{w} - \|\mathbf{v}\|^2 < 0$. *Then, the line segment connecting* $\mathbf{v}$ *and* $\mathbf{w}$ *contains a point* $\mathbf{u}$ *for which* $\|\mathbf{u}\| < \|\mathbf{v}\|$. *(See Figure 4.1.)*

Proof. Let $\mathbf{u} = \mathbf{v} + \lambda(\mathbf{w} - \mathbf{v})$, then $\mathbf{u} \in \text{conv}\{\mathbf{v}, \mathbf{w}\}$ for $0 \leq \lambda \leq 1$. Clearly, $\|\mathbf{u}\|^2 - \|\mathbf{v}\|^2 = 2\lambda\mathbf{v} \cdot (\mathbf{w} - \mathbf{v}) + \lambda^2\|\mathbf{w} - \mathbf{v}\|^2$. For $\|\mathbf{u}\|^2 - \|\mathbf{v}\|^2 = 0$, we find roots $\lambda_1 = 0$ and $\lambda_2 = -2\mathbf{v} \cdot (\mathbf{w} - \mathbf{v})/\|\mathbf{w} - \mathbf{v}\|^2$. It follows from $\mathbf{v} \cdot \mathbf{w} - \|\mathbf{v}\|^2 < 0$ that $\lambda_2 > 0$. Since $\|\mathbf{w} - \mathbf{v}\|^2 > 0$, we find that $\|\mathbf{u}\|^2 - \|\mathbf{v}\|^2$ is positive for $\lambda \to \infty$. Hence, $\|\mathbf{u}\|^2 - \|\mathbf{v}\|^2 < 0$ for $\lambda_1 < \lambda < \lambda_2$. □

Lemma 4.2. *Let* C *be a convex object and let* $\mathbf{v} \in C$ *be the point closest to the origin. Then, either* $\mathbf{v} = \mathbf{0}$ *or* $\mathbf{v} \cdot \mathbf{w} > 0$ *for all* $\mathbf{w} \in C$.

Proof. Suppose that $\mathbf{v} \neq \mathbf{0}$, and let $\mathbf{w} \in C$. Then, since C is convex, any $\mathbf{u} \in \text{conv}\{\mathbf{v}, \mathbf{w}\}$ is contained in C, and thus $\|\mathbf{u}\| \geq \|\mathbf{v}\|$. It follows from Lemma 4.1 that if for all $\mathbf{u} \in \text{conv}\{\mathbf{u}, \mathbf{w}\}$ we have $\|\mathbf{u}\| \geq \|\mathbf{v}\|$, then $\mathbf{v} \cdot \mathbf{w} - \|\mathbf{v}\|^2 \geq 0$. Hence, $\mathbf{v} \cdot \mathbf{w} \geq \|\mathbf{v}\|^2 > 0$ for all $\mathbf{w} \in C$. □

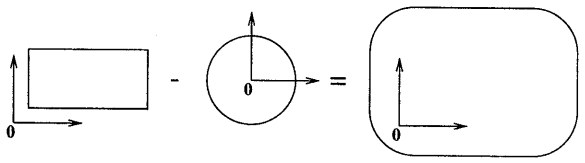


Figure 4.2: The Minkowski sum of a pair of objects

Note that the point of a convex object closest to the origin is indeed unique, since if there were two closest points, then the line segment connecting the points would contain a point that lies closer to the origin. However, since the object is convex, the line segments is contained in the object, and thus, cannot have a point closer to the origin.

The **Minkowski sum**[1] of objects A and B is defined as

$$A - B = \{\mathbf{x} - \mathbf{y} : \mathbf{x} \in A, \mathbf{y} \in B\}.$$

Figure 4.2 shows an example of the Minkowski sum of a pair of object. Although the sum of a pair of objects is a set of vectors, it is regarded as a point set. The space of this point set has the zero vector $\mathbf{0}$ as its origin. It is not very hard to show that if A and B are convex, then $A - B$ is also convex.

The **distance** between two objects A and B, denoted by $d(A, B)$, is defined as

$$d(A, B) = \min\{\|\mathbf{x} - \mathbf{y}\| : \mathbf{x} \in A, \mathbf{y} \in B\}$$

From this definition it follows that $A \cap B \neq \emptyset$ iff $d(A, B) = 0$. A pair of points $\mathbf{a} \in A, \mathbf{b} \in B$ for which $\|\mathbf{a} - \mathbf{b}\| = d(A, B)$ is called a pair of **closest points** of A and B. Notice that in general a pair of closest points of two objects is not uniquely defined.

Theorem 4.3. *Let A and B be convex objects and $\mathbf{a} \in A$ and $\mathbf{b} \in B$, a pair of closest points. Then, either $A \cap B \neq \emptyset$ or $\mathbf{a} - \mathbf{b}$ is a separating axis.*

Proof. Suppose that $A \cap B = \emptyset$. Then, $\mathbf{a} - \mathbf{b} \neq \mathbf{0}$. Clearly, $\mathbf{a} - \mathbf{b}$ is the point closest to the origin of $A - B$, the Minkowski sum of A and B. Since $A - B$ is convex, it follows from Lemma 4.2 that $(\mathbf{a} - \mathbf{b}) \cdot \mathbf{w} > 0$ for all $\mathbf{w} \in A - B$. Let $\mathbf{x} \in A$ and $\mathbf{y} \in B$. Since $(\mathbf{a} - \mathbf{b}) \cdot (\mathbf{x} - \mathbf{y}) > 0$, it follows that $(\mathbf{a} - \mathbf{b}) \cdot \mathbf{x} > (\mathbf{a} - \mathbf{b}) \cdot \mathbf{y}$. Hence, $\mathbf{a} - \mathbf{b}$ is a separating axis. □

[1] Although *Minkowski difference* seems more appropriate, we avoid using this term, since it is defined differently in many geometry texts, namely as $(A^* - B)^*$, i.e., the complement of the Minkowski sum of A's complement and B (shrink one object by the other).

Apparently, we can find a separating axis (as well as a common point) by computing a pair of closest points, which can be done using one of the distance computation algorithms described in following sections. However, other methods for finding a separating axis or a separating plane exist that often require less computations than distance algorithms. We will discuss a number of these algorithms first.

4.2.1 Separating a Pair of Polytopes

Contrary to the problem of finding a common point, the problem of finding a separating plane for polytopes can not be expressed as a linear programming problem. However, there is a less strict definition of separating plane, referred to as weak separating plane, for which the separating plane search problem *can* be expressed as an LP feasibility test. A **weak separating plane** is plane for which the objects are contained respectively in the positive and negative *closed* halfspaces. The existence of a weak separating plane does not guarantee the disjointness of the objects, but it does yield the disjointness of the interiors of the objects. The problem of finding a weak separating plane for a pair of polytopes can be expressed as an LP problem in the following way.

It can be seen that a plane that weakly separates the vertices of the polytopes, also weakly separates the polytopes themselves. For a pair of polytopes A and B, we need to find a plane $H(\mathbf{v}, \delta)$ such that for all vertices $\mathbf{a} \in \text{vert}(A)$, we have $\mathbf{v} \cdot \mathbf{a} + \delta \geq 0$, and for all vertices $\mathbf{b} \in \text{vert}(B)$, we have $\mathbf{v} \cdot \mathbf{b} + \delta \leq 0$. We see that our search space is four dimensional. This problem can be expressed as the problem of finding an $\mathbf{x} = (\mathbf{v}, \delta)$ subject to the constraints $(\mathbf{a}, 1) \cdot \mathbf{x} \geq 0$ for $\mathbf{a} \in \text{vert}(A)$, and $(\mathbf{b}, 1) \cdot \mathbf{x} \leq 0$ for $\mathbf{b} \in \text{vert}(B)$, which is clearly an LP problem. As we saw earlier, low-dimensional LP problems can be solved in time linear in the number of constraints.

Let us get back to strict separation of polytopes. The following theorem gives us a straightforward method for finding a separating axis for a pair of polytopes which can successfully be used if the number of facet orientations and edge directions is small. The proof of this theorem presented here is a shorter alternative to the one given in [45].

Theorem 4.4. *For a pair of nonintersecting polytopes, there exists a separating axis that is orthogonal to a facet of either polytope, or orthogonal to an edge from each polytope.*

Proof. Let A and B be a pair of nonintersecting polytopes. Then, $A - B$, is itself a polytope and does not contain the origin. Since a polytope can

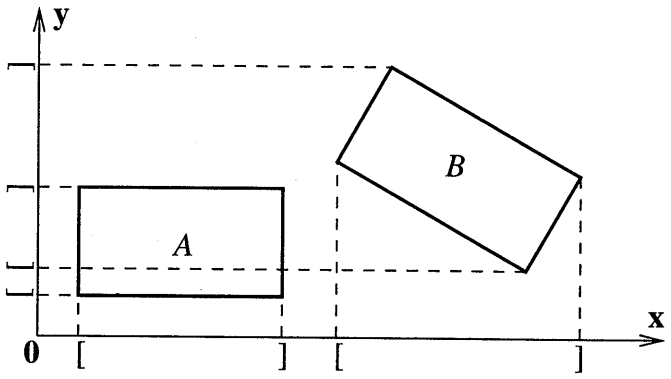


Figure 4.3: The vector **x** is a separating axis of A and B, whereas **y** is not a separating axis.

be represented as the intersection of halfspaces whose boundaries are the affine hulls of the polytope's facets [48], at least one of the facets corresponds with a halfspace that does not contain the origin. Let $H^+(\mathbf{v}, \delta)$ be such a halfspace that does not contain the origin. Since $\mathbf{0} \notin H^+(\mathbf{v}, \delta)$, we see that $\delta < 0$. For all $\mathbf{w} \in A - B$, we have $\mathbf{v} \cdot \mathbf{w} + \delta \geq 0$, and thus $\mathbf{v} \cdot \mathbf{w} > 0$, which in turn yields that $\mathbf{v}$ is a separating axis of A and B. Hence, a separating axis exists that is normal to a facet of $A - B$.

Each facet of $A - B$ is (composed of sub-facets, being) either the Minkowski sum of a facet from one polytope and a vertex from the other, or the Minkowski sum of a pair of edges from each polytope. A normal to a facet of $A - B$ is therefore either orthogonal to a face of one of the polytopes, or orthogonal to a pair of edges, one from each polytope. □

Hence, we see that a separating axis can be found by simply testing all facet orientations and all combinations of edge directions to see if one of these is a separating axis. For a pair of polytopes with f facet orientations and e edge directions each, we need to test at most $2f + e^2$ axes. If none of these axes yield a separating axis, then the polytopes must intersect.

An important example of this approach is the **separating-axes test** (SAT) for boxes, as described in [46]. A box has three facet orientations and three edge directions which results in 15 axes to be tested. A single separating axis test involves projecting both boxes onto the axis, and testing whether the projection intervals of the boxes overlap. If the intervals are disjoint, then the axis is a separating axis, as illustrated in Figure 4.3. For rectangular boxes, the axes tests can be optimized such that the 15 tests take less than 200 primitive arithmetic operations in total [46].

The same method can be applied to other polytopes as well. For instance, a triangle has one facet orientation and three edge directions. This results in a total of 11 axes to be tested for a pair of triangles. For k-DOPs, the number of possible axes will rapidly become too large for $k > 3$, in order for this method to be of any practical use. The number of facet orientations of a k-DOP is at most k. Euler's formula yields that the number of edge directions is at most $3k - 6$ [80]. Therefore, the number of axes that need to be tested for determining whether a pair of k-DOPs intersect is $2k + (3k - 6)^2$. Zachmann shows in [105] that the projection of a DOP onto an axis can be found in $O(k)$ time. Thus, an intersection test for DOPs, which involves testing all possible axes, has $O(k^3)$ time complexity. This is not particularly good, considering that a trivial common point search takes $O(k^2)$ time.

We see that for general polytopes, the number of axes is often too large for this approach to be useful. We might do better taking a heuristic approach in choosing axes to be tested as separating axes. An algorithm that takes such an approach is the Chung-Wang algorithm, which we will discuss next.

4.2.2 The Chung-Wang Algorithm

The Chung-Wang (CW) algorithm, presented in [20], is of interest to us because of its original strategy, rather than its particular utility. The CW algorithm is an iterative method for finding a weak separating axis for a pair of polytopes. A **weak separating axis** of objects A and B is a nonzero vector $\mathbf{v}$ such that

$$\mathbf{v} \cdot \mathbf{x} \geq \mathbf{v} \cdot \mathbf{y} \quad \text{for all} \quad \mathbf{x} \in A \text{ and } \mathbf{y} \in B.$$

We introduce the notion of **support mapping**, which is a function s_A that maps a vector to a point of an object A, according to

$$s_A(\mathbf{v}) \in A \quad \text{such that} \quad \mathbf{v} \cdot s_A(\mathbf{v}) = \max\{\mathbf{v} \cdot \mathbf{x} : \mathbf{x} \in A\}.$$

The value of a support mapping for a given vector is called a **support point**. Note that a support mapping of a given object may not be uniquely determined. The choice of support mapping does not matter in the applications of support mappings that we will encounter. Using this definition, we express a weak separating axis of A and B as a nonzero vector $\mathbf{v}$ for which

$$\mathbf{v} \cdot s_A(-\mathbf{v}) \geq \mathbf{v} \cdot s_B(\mathbf{v}),$$

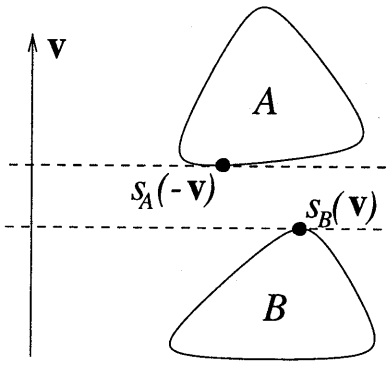


Figure 4.4: For a weak separating axis $\mathbf{v}$ we have $\mathbf{v} \cdot s_A(-\mathbf{v}) \geq \mathbf{v} \cdot s_B(\mathbf{v})$.

as illustrated in Figure 4.4. Given a polytope A, there exists a vertex $\mathbf{w}$ of A such that $\mathbf{v} \cdot \mathbf{w} = \mathbf{v} \cdot s_A(\mathbf{v})$ for any vector $\mathbf{v}$. Phrased differently, we may choose

$$s_A(\mathbf{v}) = s_{\text{vert}(A)}(\mathbf{v}),$$

i.e., for polytopes, we may restrict ourselves to support mappings that return vertices only. The naive computation of a support point for a polytope represented by a list of n vertices will take $O(n)$ time. However, it is shown in [20] that a support point can be found in $O(\log n)$ time for a polytope represented using the hierarchical representation by Dobkin and Kirkpatrick [27].

Let us express a weak separating axis of objects A and B in terms of $A - B$, the Minkowski sum of A and B. It can be shown that the mapping s_{A-B} defined by

$$s_{A-B}(\mathbf{v}) = s_A(\mathbf{v}) - s_B(-\mathbf{v})$$

is a support mapping of $A - B$. With this property we find that a nonzero vector $\mathbf{v}$, for which

$$\mathbf{v} \cdot s_{A-B}(-\mathbf{v}) \geq 0,$$

is a weak separating axis of A and B.

We will now discuss the iterative method that underlies the CW algorithm. Let $\mathbf{v}_k$ be the axis to be tested in the kth iteration. The following axis is taken as a better approximation of a possibly existing weak separating axis:

$$\mathbf{v}_{k+1} = \mathbf{v}_k - 2(\mathbf{r}_k \cdot \mathbf{v}_k)\mathbf{r}_k, \text{ where } \mathbf{r}_k = \mathbf{w}_k / \|\mathbf{w}_k\|, \text{ and } \mathbf{w}_k = s_{A-B}(-\mathbf{v}_k).$$

This choice is motivated by the observation that for a pair of spheres, for which $\mathbf{v}_k$ is not a weak separating axis, the axis $\mathbf{v}_{k+1}$ is a weak separating axis [20]. As initial axis $\mathbf{v}_0$ we may take an arbitrary unit vector. Each $\mathbf{v}_k$ has length one, since each new $\mathbf{v}_{k+1}$ is the reflection of $\mathbf{v}_k$ in the plane $H(\mathbf{r}_k, 0)$.

The CW algorithm terminates as soon as either $\mathbf{v}_k$ is a weak separating axis, i.e., $\mathbf{v}_k \cdot \mathbf{w}_k \geq 0$, or there is evidence that the objects' interiors intersect. Figure 4.5 illustrates a sequence of iterations that results in $\mathbf{v}_3$ being a separating axis.

Note that $\mathbf{r}_k$ is not defined for $\mathbf{w}_k = \mathbf{0}$. However, this case does not yield a problem, since for $\mathbf{w}_k = \mathbf{0}$, we have $\mathbf{v}_k \cdot \mathbf{w}_k = 0$, and thus, $\mathbf{v}_k$ is a weak separating axis, in which case the algorithm terminates.

Convergence is not quite proven for the CW. The following theorem expresses the strongest point that can be made.

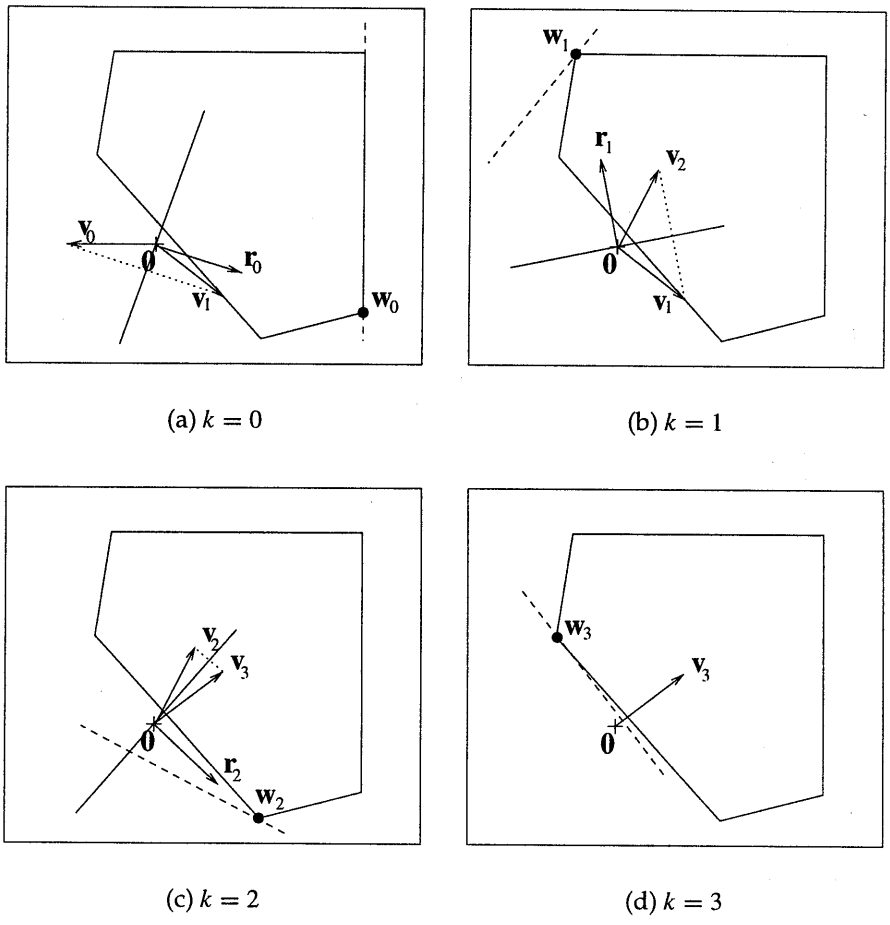
Theorem 4.5. *Suppose that unit vector* $\mathbf{u}$ *is a weak separating axis of* A *and* B, *and that* $\mathbf{v}_k$ *is not a weak separating axis. Then,* $\mathbf{v}_{k+1} \cdot \mathbf{u} \geq \mathbf{v}_k \cdot \mathbf{u}$.

Proof. We deduce, $\mathbf{v}_{k+1} \cdot \mathbf{u} = \mathbf{v}_k \cdot \mathbf{u} - 2(\mathbf{r}_k \cdot \mathbf{v}_k)(\mathbf{r}_k \cdot \mathbf{u})$. Since $\mathbf{u}$ is a weak separating axis, and $\mathbf{v}_k$ is not, we have $\mathbf{r}_k \cdot \mathbf{u} \geq 0$ and $\mathbf{r}_k \cdot \mathbf{v}_k < 0$. Hence, $\mathbf{v}_k \cdot \mathbf{u} - 2(\mathbf{r}_k \cdot \mathbf{v}_k)(\mathbf{r}_k \cdot \mathbf{u}) \geq \mathbf{v}_k \cdot \mathbf{u}$. □

Chung and Wang do claim convergence for their algorithm in [20], however the proof they give is incorrect.

Since both $\mathbf{v}_k$ and $\mathbf{v}_{k+1}$ are unit vectors, it follows from Theorem 4.5 that the angle between $\mathbf{v}_{k+1}$ and $\mathbf{u}$ is at most as large as the angle between $\mathbf{v}_k$ and $\mathbf{u}$. However, this is not sufficient to conclude that $\mathbf{v}_k \cdot \mathbf{w}_k \geq 0$, for some $k \geq 0$, i.e., it does not prove that eventually a weak separating axis is found. Further on, we will discuss how termination can be achieved for a pair of disjoint polytopes. But first, let us discuss the case where the objects are intersecting.

In the case where a weak separating axis does not exist, the algorithm terminates as soon as there is evidence that the interiors of the objects intersect. For this purpose, the CW algorithm uses a subalgorithm that tries to compute a vector $\mathbf{n}_k$ such that $\mathbf{n}_k \cdot \mathbf{w}_i \geq 0$ for all $i = 0, \ldots, k$. If such a vector does not exist, then $\mathbf{0}$ must lie in the interior of $A - B$, and thus, the interiors of A and B intersect. Chung and Wang present an $O(k)$ time algorithm for computing such an $\mathbf{n}_k$. Hence, for a pair of polytopes of respectively n and m vertices, it takes $O(k^2 + k(f(n) + f(m)))$ time to perform k iterations, where $f(n)$ is the time it takes to compute a support point for a polytope that has n vertices. Note that we may choose to perform the subalgorithm once in every fixed number of iterations, instead of



(a) $k = 0$ (b) $k = 1$

(c) $k = 2$ (d) $k = 3$

Figure 4.5: Four iterations of the CW algorithm. The dashed line segments represent the support planes $H(\mathbf{v}_k, -\mathbf{v}_k \cdot \mathbf{w}_k)$. The plane in which $\mathbf{v}_k$ is reflected is represented as a continuous line segment.

each iteration, in order to speed up the algorithm. This will however not improve the time complexity. It shows that for an acceptable performance of the algorithm the number of iterations needs to be kept small.

Discussion

So far, we have viewed the CW algorithm and some of its properties. We are now ready to discuss its merits in greater detail.

Let us explore how large the number of iterations of the CW algorithm can grow. Clearly, if $\mathbf{v}_k$ is close to being a separating axis, i.e., $\mathbf{v}_k \cdot \mathbf{w}_k \approx 0$, then $\mathbf{r}_k \cdot \mathbf{v}_k \approx 0$, and thus $\mathbf{v}_{k+1} \approx \mathbf{v}_k$, which suggests that convergence is slow. This problem will occur most prominently if the objects are (almost) touching. In our experiments, we compute a convergence factor ρ using the formula

$$\rho = \frac{\mathbf{r}_k \cdot \mathbf{v}_k}{\mathbf{r}_{k-1} \cdot \mathbf{v}_{k-1}}.$$

Here, $\mathbf{r}_k \cdot \mathbf{v}_k$ is half the length of $\mathbf{v}_{k+1} - \mathbf{v}_k$ (half the length of the dotted line in Figure 4.5). For the cases where objects were touching, we often found $\rho \approx 1$, which shows that the algorithm's convergence is extremely slow.

In general, the number of iterations can grow infinitely large. However, Chung and Wang exploit a property that enables the algorithm to terminate in a finite number of iterations. After a certain number of iterations (Chung and Wang suggest the first time a point $\mathbf{w}_k$ is returned that appeared before), the algorithm continues iterating using $\mathbf{v}_{k+1} = \mathbf{n}_k$ as a new axis. If for this axis a support point is returned that appeared before, then the axis is a weak separating axis, since $\mathbf{n}_k \cdot \mathbf{w}_i \geq 0$ for all $i = 0, \ldots, k$. Thus, in each iteration the algorithm either terminates, or the new support point $\mathbf{w}_{k+1}$ is different from all the support points returned so far. Since the Minkowski sum of two polytopes of respectively m and n vertices has at most mn vertices, at most mn different support points can be returned. Therefore, the algorithm will perform mn iterations in the worst case, before establishing a termination condition. In practice, the algorithm will often need fewer iterations, since only a small number of all the possible support points will be returned. However, for polytopes that have a lot of vertices, the number of iterations can still be quite large, which is harmful for performance, considering the algorithm's $O(k^2)$ time complexity.

Further iterations can be saved by exploiting frame coherence. We saw that the initial axis $\mathbf{v}_0$ may be chosen arbitrarily. In cases were there is a lot of frame coherence, an existing separating axis of the object pair from a previous frame is likely to be a separating axis in the current frame. If

we take this axis as initial axis, the algorithm will often need only one iteration.

At first glance, the CW algorithms seems a good candidate for generalization to other convex objects besides polytopes, since we can also provide support mappings for non-polytopes. However, since the number of support points of a non-polytope is unbounded, we cannot guarantee that the algorithm terminates. Due to extremely slow convergence for cases where the objects are almost touching, this termination condition is of crucial importance; it is not merely a safety in case of numerical instability as stated in the article [20].

Despite Snepvangers' efforts to generalize the algorithm for application on general convex objects [89], certain configurations of objects exist that require arbitrary many iterations. The CW algorithm does not appear to be applicable to objects other than polytopes. In this respect, the Gilbert-Johnson-Keerthi algorithm, which we will discuss in Section 4.4, is a similar iterative method that can be generalized more successfully to general convex objects.

4.3 Computing the Distance

The problem of computing the distance between a pair of objects is more general than the problem of detecting whether the objects intersect, since two objects intersect iff their distance is zero. A witness of the distance, a pair of closest points, gives us a common point if the objects intersect, and a separating axis (the difference of the closest points) in case the objects are disjoint. Hence, algorithms for computing the distance are useful in the context of collision detection. In this section, we present an overview of algorithms for computing the distance between polytopes.

One of the first significant solutions to the polytope distance computation problem was presented by Dobkin and Kirkpatrick in [27]. They devised an algorithm for computing the distance between two polytopes in time linear in the total number of vertices. This algorithm utilizes a hierarchical representation for two- and three-dimensional polytopes. They later showed in [28] that the distance between a pair of polytopes with respectively m and n vertices can be found in $O(\log m \log n)$ time using the same hierarchical representation.

The distance between two objects A and B can be expressed in terms of their Minkowski sum $A - B$ as

$$d(A, B) = \|v(A - B)\|,$$

where $v(C)$ is defined as the point in C nearest to the origin, i.e.,

$$v(C) \in C \quad \text{and} \quad \|v(C)\| = \min\{\|\mathbf{x}\| : \mathbf{x} \in C\}.$$

An approach that can be taken is the following. First, an explicit representation of the Minkowski sum $A - B$ is computed, after which the point $v(A - B)$ is computed using this representation. We saw that the Minkowski sum of two polytopes is itself a polytope, and can thus be finitely represented using any of the discussed polytope representations. Cameron and Culley used this approach in an algorithm for computing the distance, in which they represented $A - B$ as the intersection of a set of halfspaces [14].

It can be seen that this approach will not yield a particularly low time complexity, since for a pair of polytopes A and B of respectively m and n vertices, an explicit representation of $A - B$ may require $O(mn)$ space, and thus the best construction algorithm for $A - B$ takes $O(mn)$ time. In order to improve this bound we need to take an approach in which $A - B$ is not explicitly constructed. A nice example of such an approach applied to convex polygons in the plane is presented in [82]. In Section 4.4 we will describe the GJK distance algorithm, which takes this approach for computing $v(A - B)$ of convex objects A and B in three-dimensional space.

The Lin-Canny algorithm (LC) is an incremental algorithm for computing a pair of closest features of convex polyhedra [61]. A feature is a vertex, an edge, or a facet of the boundary of a polyhedron. LC forms the heart of I-COLLIDE, a publicly available software library for interactive collision detection [22]. LC's utility for interactive collision detection follows from its ability to compute a pair of closest features in near constant time if the closest features are approximately known. This is useful when there is a lot of frame coherence, as commonly is the case in computer animation. Since then, the closest features from a previous frame are likely to be approximate to the closest features in the current frame.

LC finds the closest features by iteratively 'walking' across the boundary of the polyhedra towards the features of the boundary that lie closer to each other. The algorithm starts at an arbitrary pair of features, preferably features that lie near to the closest features. In each iteration, the algorithm proceeds to a neighboring pair of features that lie closer to each other than the previous pair.

Special care should be taken in handling local minima, i.e., feature pairs that are not closest, and for which no closer neighboring feature pair exists. The existence of a local minimum may be the result of the polyhedra interpenetrating, however, it does not guarantee this. Additional computations are necessary in order to determine if the polyhedra truly intersect.

	DK [28]	CC [14]	LC [61]
Representation	hierar. repr.	winged-edge	winged-edge
Space bound	$O(n)$	$O(n^2)$	$O(n)$
Time bound	$O(\log^2 n)$	$O(n^2)$	empirically $O(n)$ ($O(1)$, incremental)
Implementation	unknown	exists [14]	I-COLLIDE [22], V-Clip [66]

Table 4.2: Distance computation algorithms

In theory, the running time of the LC algorithm has a worst-case upper bound of $O(n^2)$ for a pair of polyhedra of n vertices each, since there are $O(n^2)$ feature pairs. However, Lin and Canny claim that empirically their algorithm has a time bound that is linear in the number of vertices, if no special initialization is done, and constant, when the algorithm is initialized by a pair of features that lie near to the closest features.

It has been noted by Mirtich that the original LC suffers from cycling, i.e., infinitely alternating of pairs of features, for degenerate configurations of nonintersecting objects [66]. He presents a variant of LC, called V-Clip, which is claimed to solve this problem and has better performance than the original algorithm. An implementation by Mirtich of the V-Clip algorithm is made publicly available as the V-Clip collision detection library.

Table 4.2 shows an overview of the discussed distance computation algorithms. Although the Dobkin-Kirkpatrick algorithm has the best worst-case time bound, we cannot draw conclusions regarding its actual performance, since there is no reference to an implementation of this algorithm. Of the mentioned distance algorithms, the LC-based algorithms are best suited for application in computer animation, where there is usually a lot of frame coherence. In the following section we will discuss the Gilbert-Johnson-Keerthi distance algorithm, which can be tailored to incrementally compute the distance in expected constant time, as demonstrated by Cameron in [13]. However, we found it to be more useful when applied to incremental separating-axis computation.

4.4 The Gilbert-Johnson-Keerthi Algorithm

In this section we discuss the Gilbert-Johnson-Keerthi algorithm (GJK), an iterative method for computing the distance between convex objects. The original GJK distance algorithms is applicable to polytopes only [40]. Later

on, Gilbert and Foo presented an extended GJK algorithm to be used for convex objects in general [39].

We present a GJK implementation that has improved performance, robustness, and versatility over earlier implementations. The performance improvements are the result of: (a) data caching and smarter selection of sub-simplices in the GJK subalgorithm, (b) early termination on finding a separating axis, in the application of GJK to intersection detection, and (c) exploitation of frame coherence by separating axis caching. Regarding robustness, we present a solution to a termination problem in the original GJK due to rounding errors, which was noted by Nagle [71]. Finally, regarding versatility, we show how GJK can be applied to a large family of geometric primitives, which includes boxes, spheres, cones and cylinders, and their images under affine transformation, thus demonstrating the usefulness of GJK for collision detection of objects described in VRML [8].

4.4.1 Overview of GJK

This section describes the extended GJK for general convex objects, first presented in [39].

GJK is essentially a descent method for approximating $v(A - B)$ for convex A and B. In each iteration a simplex is constructed that is contained in $A - B$ and lies nearer to the origin than the simplex constructed in the previous iteration. We define W_k as the set of vertices of the simplex constructed in the k-th iteration ($k \geq 1$), and $\mathbf{v}_k$ as $v(\text{conv}(W_k))$, the point in the simplex nearest to the origin. Initially, we take $W_0 = \emptyset$, and $\mathbf{v}_0$, an arbitrary point in $A - B$. Since $A - B$ is convex and $W_k \subseteq A - B$, we see that $\mathbf{v}_k \in A - B$, and thus $\|\mathbf{v}_k\| \geq \|v(A - B)\|$ for all $k \geq 0$.

GJK generates the sequence of simplices in the following way. Let $\mathbf{w}_k = s_{A-B}(-\mathbf{v}_k)$, where s_{A-B} is a support mapping of $A - B$. We take $\mathbf{v}_{k+1} = v(\text{conv}(W_k \cup \{\mathbf{w}_k\}))$, and as W_{k+1} we take the smallest set $X \subseteq W_k \cup \{\mathbf{w}_k\}$, such that $\mathbf{v}_{k+1}$ is contained in $\text{conv}(X)$. It can be seen that exactly one such X exists, and that it must be affinely independent. Figure 4.6 illustrates a sequence of iterations of the GJK algorithm in two dimensions.

In order to proof that the sequence $\{\mathbf{v}_k\}$ converges to $v(A - B)$, we require the following theorem.

Theorem 4.6. *For $\mathbf{v}_k \in A - B$, we have $\|\mathbf{v}_{k+1}\| \leq \|\mathbf{v}_k\|$, with equality only if $\mathbf{v}_k = v(A - B)$.*

Proof. Let $\mathbf{v}_k \in A - B$. Then, $\|\mathbf{v}_{k+1}\| = \min\{\|\mathbf{x}\| : \mathbf{x} \in \text{conv}(W_k \cup \{\mathbf{w}_k\})\} \leq \|\mathbf{v}_k\|$, since $\mathbf{v}_k \in \text{conv}(W_k)$. Furthermore, $\mathbf{v}_k \cdot \mathbf{w}_k - \|\mathbf{v}_k\|^2 \leq 0$, with equality

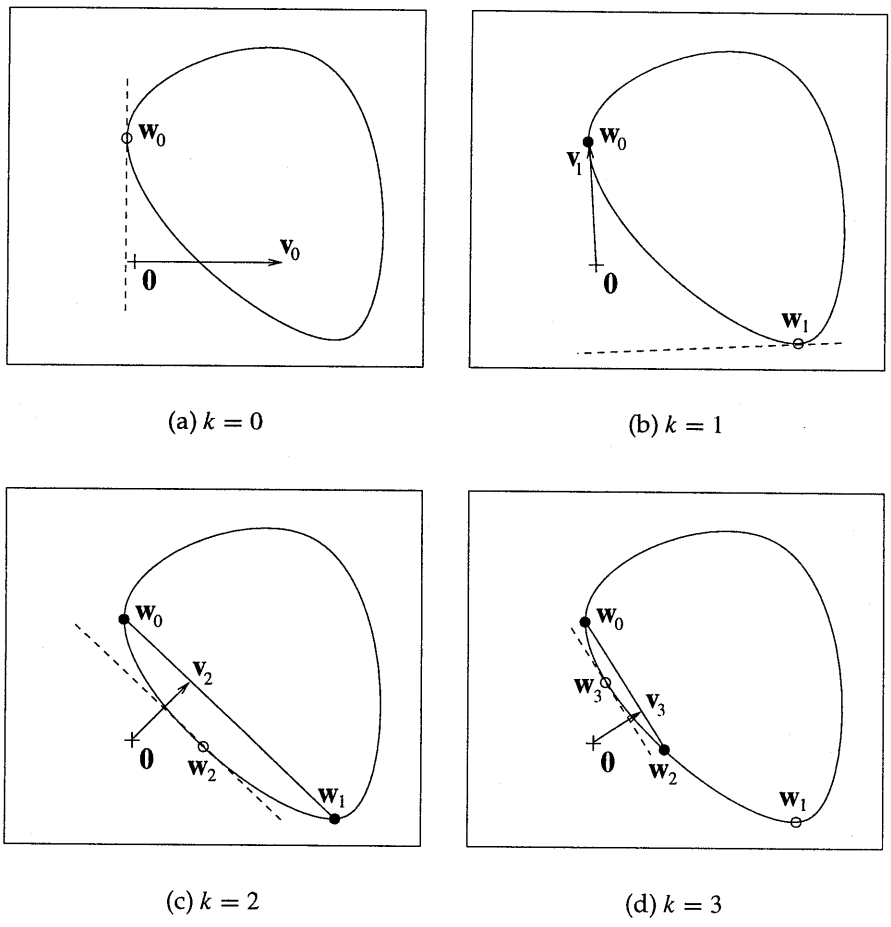


Figure 4.6: Four iterations of the GJK algorithm. The dashed lines represent the support planes $H(-\mathbf{v}_k, \mathbf{v}_k \cdot \mathbf{w}_k)$. The points of W_k are drawn in black.

only if $\mathbf{v}_k = v(A-B)$ [40]. Suppose $\mathbf{v}_k \neq v(A-B)$. Then, $\mathbf{v}_k \cdot \mathbf{w}_k - \|\mathbf{v}_k\|^2 < 0$, and thus according to Lemma 4.1, there exists a $\mathbf{u} \in \text{conv}\{\mathbf{v}_k, \mathbf{w}_k\}$ such that $\|\mathbf{u}\| < \|\mathbf{v}_k\|$. Since $\mathbf{v}_k \in \text{conv}(W_k)$, and thus $\text{conv}\{\mathbf{v}_k, \mathbf{w}_k\}$ is contained in $\text{conv}(W_k \cup \{\mathbf{w}_k\})$, we find that $\|\mathbf{v}_{k+1}\| < \|\mathbf{v}_k\|$. □

Theorem 4.6 provides a necessary, yet not sufficient condition for global convergence. In order to prove that $\mathbf{v}_k$ indeed converges to $v(A-B)$, we must further show that the mapping of $\mathbf{v}_k$ to $\mathbf{v}_{k+1}$ is **closed** [62]. We refer to [39] for a proof of closeness of GJK.

For polytopes, GJK arrives at $\mathbf{v}_k = v(A-B)$ in a finite number of iterations, as shown in [40]. For non-polytopes this may not be the case. For these type of objects, it is necessary that the algorithm terminates as soon as $\mathbf{v}_k$ lies within a given tolerance from $v(A-B)$. The error of $\mathbf{v}_k$ is estimated by maintaining a lower bound for $\|v(A-B)\|$. As a lower bound we may take the signed distance from the origin to the supporting plane $H(-\mathbf{v}_k, \mathbf{v}_k \cdot \mathbf{w}_k)$, which is

$$\delta_k = \mathbf{v}_k \cdot \mathbf{w}_k / \|\mathbf{v}_k\|.$$

This is a proper lower bound since for positive δ_k, the origin lies in the positive halfspace, whereas $A-B$ is contained in the negative halfspace of the plane. In Figure 4.6 we see that δ_k is positive in the cases where the dashed line crosses the arrow.

Let f be a function on $A-B$ defined by $f(\mathbf{v}) = \mathbf{v} \cdot s_{A-B}(-\mathbf{v})$. Then, $\delta_k = f(\mathbf{v}_k)/\|\mathbf{v}_k\|$. Since f is continuous, and $f(v(A-B)) = \|v(A-B)\|^2$, it can be seen that $\{\delta_k\}$ converges to $\|v(A-B)\|$. Hence, the algorithm terminates in a finite number of iterations for any positive error tolerance.

However, contrary to $\|\mathbf{v}_k\|$, the lower bound δ_k may not be monotonous in k, i.e., it is possible that $\delta_j < \delta_i$ for $j > i$. Furthermore, 0 is a trivial lower bound for $\|v(A-B)\|$. Therefore, we use

$$\mu_k = \max\{0, \delta_0, \ldots, \delta_k\}$$

as a lower bound, which is often tighter than δ_k. A monotonous lower bound is needed for the following reason. For certain configurations of objects (in particular objects that have flat boundary elements), the function f is **ill-conditioned**, i.e., a small change in $\mathbf{v}$ may result in a large change in $f(\mathbf{v})$. Since the computation of $\mathbf{v}_k$ with finite precision arithmetics inevitably suffers from rounding errors, the computed value for δ_k may be considerably smaller than its actual value. The relative error in $\mathbf{v}_k$ is larger for sets W_k that are close to being affinely dependent, as we will see Subsection 4.4.4. GJK has a tendency to generate simplices that are

progressively more oblong, i.e., closer to being affinely dependent, as the number of iterations increases. Hence, for large k, the computed values for δ_k may be less reliable.

Given ε, a tolerance for the absolute error in $\|\mathbf{v}_k\|$, the algorithm terminates as soon as $\|\mathbf{v}_k\| - \mu_k \leq \varepsilon$. Algorithm 4.1 described the GJK distance algorithm presented in pseudo-code.

Algorithm 4.1 The GJK distance algorithm

```
v := "arbitrary point in A − B";
W := ∅;
μ := 0;
close_enough := false;
while not close_enough and v ≠ 0 do begin
  w := s_{A−B}(−v);
  δ := v · w/‖v‖;
  μ := max{μ, δ};
  close_enough := ‖v‖ − μ ≤ ε;
  if not close_enough then begin
    v := υ(conv(W ∪ {w}));
    W := "smallest X ⊆ W ∪ {w} such that v ∈ conv(X)";
  end
end;
return ‖v‖
```

We now focus on the computation of $\mathbf{v} = \upsilon(\text{conv}(Y))$ for an affinely independent set Y and the determination of the smallest $X \subseteq Y$ such that $\mathbf{v} \in \text{conv}(X)$. These operations are performed by a single subalgorithm. The requested subset $X = \{\mathbf{x}_0, \ldots, \mathbf{x}_n\}$ of Y is characterized by

$$\mathbf{v} = \sum_{i=0}^{n} \lambda_i \mathbf{x}_i \quad \text{where} \quad \sum_{i=0}^{n} \lambda_i = 1 \quad \text{and} \quad \lambda_i > 0.$$

This subset X is the largest of all nonempty $Z \subseteq Y$ for which all the λ_i-s of the point $\upsilon(\text{aff}(Z))$ are positive. The requested point $\mathbf{v}$ is the point $\upsilon(\text{aff}(X))$.

It remains to explain how to find the λ_i-s for $\upsilon(\text{aff}(X))$, where X is affinely independent. We observe that the vector $\mathbf{v} = \upsilon(\text{aff}(X))$ is perpendicular to $\text{aff}(X)$, i.e., $\mathbf{v} \in \text{aff}(X)$ and $\mathbf{v} \cdot (\mathbf{x}_i - \mathbf{x}_0) = 0$ for $i = 1, \ldots, n$.

Hence, the λ_i-s are found by solving a system of linear equations. We apply Cramer's rule to solve these systems of equations. Since we need to find solutions for all nonempty subsets of Y, we exploit the recursion in Cramer's rule. Let $Y = \{\mathbf{y}_0, \ldots, \mathbf{y}_n\}$, where $n < 4$. Each nonempty $X \subseteq Y$ is identified by a nonempty $I_X \subseteq \{0, \ldots, n\}$ such that $X = \{\mathbf{y}_i : i \in I_X\}$. We obtain the following recursively defined solutions. For each subset X, we have $\lambda_i = \Delta_i(X)/\Delta(X)$, where $\Delta(X) = \sum_{i \in I_X} \Delta_i(X)$, and

$$\begin{aligned} \Delta_i(\{\mathbf{y}_i\}) &= 1 \\ \Delta_j(X \cup \{\mathbf{y}_j\}) &= \sum_{i \in I_X} \Delta_i(X)(\mathbf{y}_i \cdot \mathbf{y}_k - \mathbf{y}_i \cdot \mathbf{y}_j), \end{aligned}$$

where $j \notin I_X$ and k is an arbitrary but fixed member of I_X, for instance $k = \min(I_X)$. The smallest $X \subseteq Y$ such that $\mathbf{v} \in \text{conv}(X)$ can now be characterized as the subset X for which (i) $\Delta_i(X) > 0$ for each $i \in I_X$, and (ii) $\Delta_j(X \cup \{\mathbf{y}_j\}) \leq 0$, for all $j \notin I_X$. The subalgorithm successively tests each nonempty subset X of Y until it finds one for which (i) and (ii) hold.

Finally, a pair of closest points is computed as follows. At termination, we have a representation of $\mathbf{v} \approx v(A - B)$ as

$$\mathbf{v} = \sum_{i=0}^{n} \lambda_i \mathbf{y}_i \quad \text{where} \quad \sum_{i=0}^{n} \lambda_i = 1 \quad \text{and} \quad \lambda_i > 0.$$

Each $\mathbf{y}_i = \mathbf{p}_i - \mathbf{q}_i$, where $\mathbf{p}_i$ and $\mathbf{q}_i$ are support points of respectively A and B. Let $\mathbf{a} = \sum_{i=0}^{n} \lambda_i \mathbf{p}_i$ and $\mathbf{b} = \sum_{i=0}^{n} \lambda_i \mathbf{q}_i$. Since A and B are convex, it is clear that $\mathbf{a} \in A$ and $\mathbf{b} \in B$. Furthermore, it can be seen that $\mathbf{a} - \mathbf{b} = \mathbf{v}$. Hence, $\mathbf{a}$ and $\mathbf{b}$ are closest points of A and B.

4.4.2 Support Mappings

In order to use GJK on a given class of objects, all we need is a support mapping for that class. In this section we discuss the computation of the support points for a number of geometric primitives and their images under affine transformation. Using the support mappings presented here, the GJK algorithm can be applied to distance computation and collision detection between objects described in VRML [8].

Polytope

The set of polytopes includes simplices (points, line segments, triangles, and tetrahedra), convex polygons, and convex polyhedra. For a polytope

A, we may take $s_A(\mathbf{v}) = s_{\text{vert}(A)}(\mathbf{v})$, i.e.,

$$s_A(\mathbf{v}) \in \text{vert}(A) \quad \text{where} \quad \mathbf{v} \cdot s_A(\mathbf{v}) = \max\{\mathbf{v} \cdot \mathbf{x} : \mathbf{x} \in \text{vert}(A)\}.$$

Obviously, a support point of a polytope can be computed in linear time with respect to the number of vertices of the polytope. However, it has been mentioned in a number of publications [14, 20, 13, 75] that by exploiting frame coherence, the cost of computing a support point of a convex polyhedron can be reduced to almost constant time. For this purpose, an adjacency graph of the vertices is maintained with each polytope. Each edge on the polytope is an edge in the graph. In this way, a support point that lies close to the previously returned support point can be found much faster using local search. This technique is commonly referred to as **hill climbing**. The adjacency graph of polytope vertices can be obtained by computing the convex hull, for instance, using *Qhull* [5].

Box

A Box primitive is a rectangular parallelepiped centered at the origin and aligned with the coordinate axes. Let A be a Box with extents $2\eta_x$, $2\eta_y$, and $2\eta_y$. Then, we take as support mapping for A,

$$s_A((x, y, z)^{\mathrm{T}}) = (sgn(x)\eta_x, sgn(y)\eta_y, sgn(z)\eta_z)^{\mathrm{T}},$$

where $sgn(x) = -1$, if $x < 0$, and 1, otherwise.

Sphere

A Sphere primitive is a ball centered at the origin. The support mapping of a Sphere A with radius ρ is

$$s_A(\mathbf{v}) = \begin{cases} \frac{\rho}{\|\mathbf{v}\|}\mathbf{v} & \text{if } \mathbf{v} \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Cone

A Cone primitive is a capped cone that is centered at the origin and whose central axis is aligned with the y-axis. Let A be a Cone with a radius of ρ at its base, and with its apex at $y = \eta$ and its base at $y = -\eta$. Then, the for the top angle α we have $\sin(\alpha) = \rho/\sqrt{\rho^2 + (2\eta)^2}$. Let $\sigma = \sqrt{x^2 + z^2}$, the

distance from $(x, y, z)^T$ to the y-axis. We choose as support mapping for A, the mapping

$$s_A((x, y, z)^T) = \begin{cases} (0, \eta, 0)^T & \text{if } y > \|(x, y, z)^T\| \sin(\alpha) \\ (\frac{\rho}{\sigma}x, -\eta, \frac{\rho}{\sigma}z)^T & \text{else, if } \sigma > 0 \\ (0, -\eta, 0)^T & \text{otherwise.} \end{cases}$$

Cylinder

A Cylinder primitive is a capped cylinder that again is centered at the origin and whose central axis is aligned with the y-axis. Let A be a Cylinder with a radius of ρ, and with its top at $y = \eta$ and its bottom at $y = -\eta$. We find as support mapping for A the mapping

$$s_A((x, y, z)^T) = \begin{cases} (\frac{\rho}{\sigma}x, sgn(y)\eta, \frac{\rho}{\sigma}z)^T & \text{if } \sigma > 0 \\ (0, sgn(y)\eta, 0)^T & \text{otherwise.} \end{cases}$$

Affine Transformation

Given a class of objects for which we have a support mapping, the following theorem yields a method for computing support points for images under affine transformations of objects of this class.

Theorem 4.7. *Given s_A, a support mapping of object A, and $\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$, an affine transformation, a support mapping for $\mathbf{T}(A)$, the image of A under $\mathbf{T}$, is*

$$s_{\mathbf{T}(A)}(\mathbf{v}) = \mathbf{T}(s_A(\mathbf{B}^T\mathbf{v})).$$

Proof. A support mapping $s_{\mathbf{T}(A)}$ is characterized by

$$\mathbf{v} \cdot s_{\mathbf{T}(A)}(\mathbf{v}) = \max\{\mathbf{v} \cdot \mathbf{T}(\mathbf{x}) : \mathbf{x} \in A\}.$$

We rewrite the right member of this equation using the following deduction.

$$\mathbf{v} \cdot \mathbf{T}(\mathbf{x}) = \mathbf{v} \cdot \mathbf{B}\mathbf{x} + \mathbf{v} \cdot \mathbf{c} = \mathbf{v}^T\mathbf{B}\mathbf{x} + \mathbf{v} \cdot \mathbf{c} = (\mathbf{B}^T\mathbf{v})^T\mathbf{x} + \mathbf{v} \cdot \mathbf{c} = (\mathbf{B}^T\mathbf{v}) \cdot \mathbf{x} + \mathbf{v} \cdot \mathbf{c}.$$

This equation is used in the steps marked by $(*)$ in the following deduction.

$$\begin{aligned} \max\{\mathbf{v} \cdot \mathbf{T}(\mathbf{x}) : \mathbf{x} \in A\} &\overset{(*)}{=} \max\{(\mathbf{B}^T\mathbf{v}) \cdot \mathbf{x} + \mathbf{v} \cdot \mathbf{c} : \mathbf{x} \in A\} \\ &= \max\{(\mathbf{B}^T\mathbf{v}) \cdot \mathbf{x} : \mathbf{x} \in A\} + \mathbf{v} \cdot \mathbf{c} \\ &= (\mathbf{B}^T\mathbf{v}) \cdot s_A(\mathbf{B}^T\mathbf{v}) + \mathbf{v} \cdot \mathbf{c} \\ &\overset{(*)}{=} \mathbf{v} \cdot \mathbf{T}(s_A(\mathbf{B}^T\mathbf{v})) \end{aligned}$$

Hence, $s_{\mathbf{T}(A)}(\mathbf{v}) = \mathbf{T}(s_A(\mathbf{B}^T\mathbf{v}))$ is a support mapping of $\mathbf{T}(A)$. □

Note that the inverse of $\mathbf{B}$ is not used for computing a support point of an affinely transformed object, since for support mappings the vector $\mathbf{v}$ acts as a normal, rather than the difference of two points.

4.4.3 Improving Speed

This section presents a fast implementation of the subalgorithm and a collision detection algorithm derived from the GJK distance algorithm.

Subalgorithm

It is hinted in [40] that by caching and reusing results of dot products, substantial performance improvement can be obtained. Since some or all vertices in W_k reappear in W_{k+1}, many dot products from the k-th iteration are also needed in the $k+1$-th iteration. We will show how this caching of dot products is implemented efficiently.

In order to minimize the caching overhead, we assign an index number to each new support point, which is invariant for the duration that the support point is a member of $W_k \cup \{\mathbf{w}_k\}$. Since $W_k \cup \{\mathbf{w}_k\}$ has at most four points, and each point that is discarded will not reappear, we need to cache data for only four points. The support points are stored in an array $\mathbf{y}$ of length four. The index of each support point is its array index. The set W_k is identified by a subset of $\{0, 1, 2, 3\}$, which is implemented as a bit-array b, i.e., $W_k = \{\mathbf{y}[i] : b[i] = 1, i = 0, 1, 2, 3\}$. The index number of the new support point $\mathbf{w}_k$ is the smallest i for which $b[i] = 0$. Note that a free 'slot' for $\mathbf{w}_k$ is always available during iterations, because if W_k has four elements, then $\mathbf{v}_k = v(\text{conv}(W_k))$ must be zero, since W_k is affinely independent, in which case the algorithm terminates immediately without computing a support point.

The dot products of all pairs $\mathbf{y}[i], \mathbf{y}[j] \in W_k \cup \{\mathbf{w}_k\}$ are stored in a 4×4 array d, i.e., $d[i, j] = \mathbf{y}[i] \cdot \mathbf{y}[j]$. In each iteration, we need to compute the dot products of the pairs containing $\mathbf{w}_k$ only. The other dot products are already computed in previous iterations. For a W_k containing n points, this takes $n + 1$ dot product computations.

We further improve the performance of the subalgorithm by caching the values of the $\Delta_i(X)$-s. Let $Y = W_k \cup \{\mathbf{w}_k\}$. For many of the $X \subseteq Y$, the $\Delta_i(X)$-s are needed in several iterations, and are therefore better cached and reused instead of recomputed. For this purpose, each subset X is identified by the integer value of the corresponding bit-array. For instance, for $X = \{\mathbf{y}[0], \mathbf{y}[3]\}$, we find bit-array 1001, corresponding to integer value

$2^0 + 2^3 = 9$. The values of $\Delta_i(X)$ for each subset X are stored in a 16×4 array D. The element $D[x, i]$ stores the value of $\Delta_i(X)$, where x is the integer value corresponding with subset X. Only the elements $D[x, i]$ for which bit i of bit-array x is set, are used. Similar to the dot product computations, we only need to compute, in each k-th iteration, the values of $\Delta_i(X)$ for the subsets X containing the new support point $\mathbf{w}_k$, since the other values are computed in previous iterations.

Another improvement concerning the subalgorithm is based on the following theorem.

Theorem 4.8. *For each k-th iteration, where $k \geq 1$, we have $\mathbf{w}_k \in W_{k+1}$.*

Proof. Suppose $\mathbf{w}_k \notin W_{k+1}$, then $v(W_{k+1}) = v(W_k)$, and thus $\mathbf{v}_{k+1} = \mathbf{v}_k$. But since $\|\mathbf{v}_{k+1}\| < \|\mathbf{v}_k\|$, this yields a contradiction. □

Consequently, only those subsets X, for which $\mathbf{w}_k \in X$ need to be tested by the subalgorithm. This reduces the number of subsets from $2^{n+1} - 1$ to 2^n, where n is the number of elements in W_k.

Collision Detection

For deciding whether two objects intersect, we do not need to have the distance between them. We merely need to know whether the distance is equal to zero or not. Hence, as soon as the lower bound for the distance becomes positive, the algorithm may terminate returning a nonintersection. The lower bound is positive iff $\mathbf{v}_k \cdot \mathbf{w}_k > 0$, i.e., $\mathbf{v}_k$ is a **separating axis** of A and B. In general, GJK needs less iterations for finding a separating axis of a pair of nonintersecting objects than for computing an accurate approximation of $v(A - B)$. For instance in Figure 4.6, the vector $\mathbf{v}_k$ is a separating axis for the first time when $k = 2$. If the objects intersect, then the algorithm terminates on $\mathbf{v}_k = \mathbf{0}$, returning an intersection. Algorithm 4.2 shows an algorithm for computing a separating axis, which is derived from the GJK distance algorithm. Besides requiring fewer iterations in case of nonintersecting objects, the collision detection algorithm performs better than the distance algorithm for another reason. Notice that the value of $\|\mathbf{v}\|$ is not needed in the GJK separating axis algorithm. The computation of $\|\mathbf{v}\|$ involves evaluating a square root, which is an expensive operation. A single iteration of the collision detection algorithm is therefore significantly cheaper than an iteration of the distance algorithm.

Also, note that in the collision detection algorithm, $\mathbf{v}$ does not need to be initialized by a point in $A - B$, since the length of $\mathbf{v}$ does not matter. This feature is convenient for exploiting frame coherence.

Algorithm 4.2 The GJK separating axis algorithm

$\mathbf{v} :=$ "arbitrary vector";
$W := \emptyset$;
repeat
  $\mathbf{w} := s_{A-B}(-\mathbf{v})$;
  if $\mathbf{v} \cdot \mathbf{w} > 0$ then return false;
  $\mathbf{v} := v(\text{conv}(W \cup \{\mathbf{w}\}))$;
  $W :=$ "smallest $X \subseteq W \cup \{\mathbf{w}\}$ such that $\mathbf{v} \in \text{conv}(X)$";
until $\mathbf{v} = \mathbf{0}$;
return true

Similar to closest-point tracking algorithms, such as the Lin-Canny closest feature algorithm [61], and Cameron's Enhanced GJK algorithm [13], an incremental version of the GJK separating axis algorithm shows almost constant time complexity per frame for convex objects of arbitrary complexity, if frame coherence is high. The incremental separating axis GJK algorithm, further referred to as ISA-GJK, exploits frame coherence by using the separating axis from the previous frame as initial vector for $\mathbf{v}$. If the degree of coherence between frames is high, then the separating axis from the previous frame is likely to be a separating axis in the current frame, in which case ISA-GJK terminates in the first iteration. Figure 4.7 shows the behavior of ISA-GJK for a smoothly moving object. We saw in Section 4.4.2 that a support point can be computed in constant time for quadrics and, if coherence is high, in nearly constant time for arbitrary polytopes. Hence, in these cases, ISA-GJK takes nearly constant time per frame.

In order to compare the performance of ISA-GJK with existing algorithms, we conducted the following experiment. As benchmark we took the multi-body simulation from I-COLLIDE [22]. This is a simulation of a number of polyhedra that move freely inside a cubic space. The number, complexity, density, and translational and rotational velocities of the objects in the space can be varied in order to test the algorithms under different settings. The simulation has a simple type of collision response. It exchanges the translational velocities of each colliding pair of objects, thus simulating a pseudo-elastic reaction. Objects also bounce off the walls of the cubic space in order to constrain them inside the space.

Using this benchmark, we compared the performance of ISA-GJK to Lin-Canny's and Chung-Wang's. For testing Lin-Canny we used the orig-

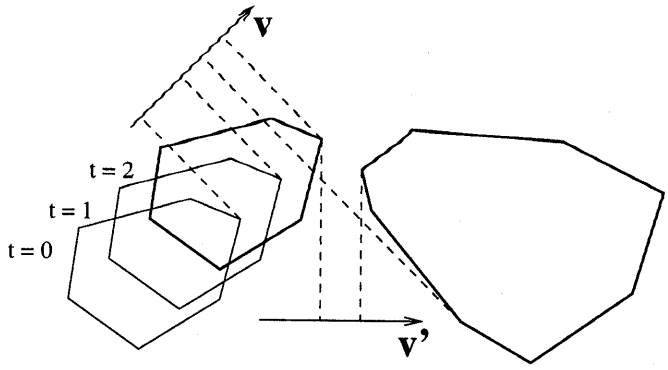


Figure 4.7: Incremental separating axis computation using ISA-GJK. The separating axis $\mathbf{v}$ from $t = 0$ is also a separating axis for $t = 1$. However, $\mathbf{v}$ fails to be a separating axis for $t = 2$. A new separating axis $\mathbf{v}'$ is computed using $\mathbf{v}$ as initial axis.

inal I-COLLIDE [21]. For Chung-Wang we took Q-COLLIDE, a version of I-COLLIDE in which the Lin-Canny closest feature test is replaced by the Chung-Wang intersection test. Unfortunately, Q-COLLIDE is no longer publicly available, however, we managed to obtain the source code before it was removed from its web site [19]. Finally, for testing ISA-GJK, we replaced the Lin-Canny test in I-COLLIDE by a C++ implementation of our ISA-GJK intersection test.

The tests were performed on a Sun UltraSPARC-I (167MHz), compiled using the GNU C/C++ compiler with '-O2' optimization. As default setting we used 20 objects, each having 20 vertices. The default density was set at 5% of the space being occupied by objects. The translational velocity of an object is expressed in the percentage of its radius the object is displaced in each frame. The default value is 5%. The default rotational velocity is 10 degrees per frame. For each setting, we measured the times for the three algorithms by simulating 50,000 frames.

We experimented with different densities, translational velocities, and rotational velocities. The results of this experiment are shown in Figure 4.8, 4.9, and 4.10. Overall, ISA-GJK is roughly five times as fast as Lin-Canny. However, with respect to Chung-Wang, ISA-GJK takes, on average, twice as much time. We did not found significant differences in accuracy for these settings. ISA-GJK and Chung-Wang consistently return the same collisions, whereas Lin-Canny occasionally misses a collision detected by the other two, although the differences are minimal.

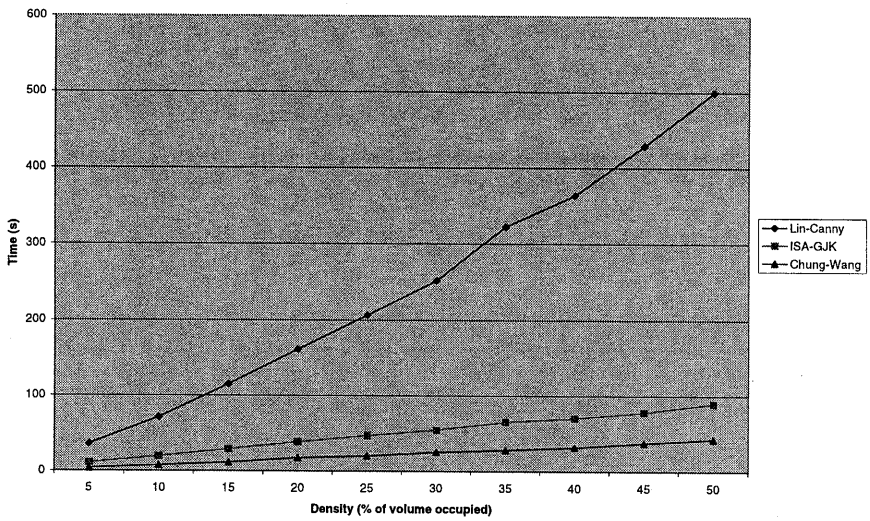


Figure 4.8: Performance under density variations

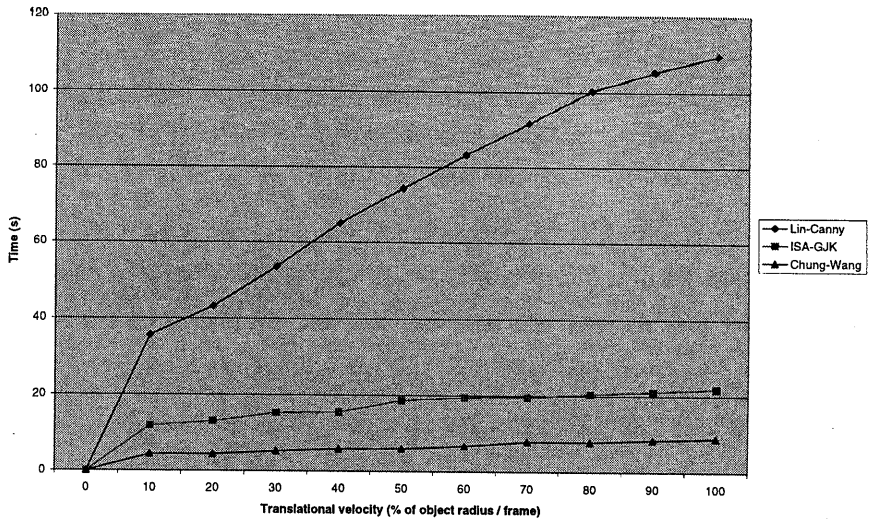


Figure 4.9: Performance under translational-velocity variations

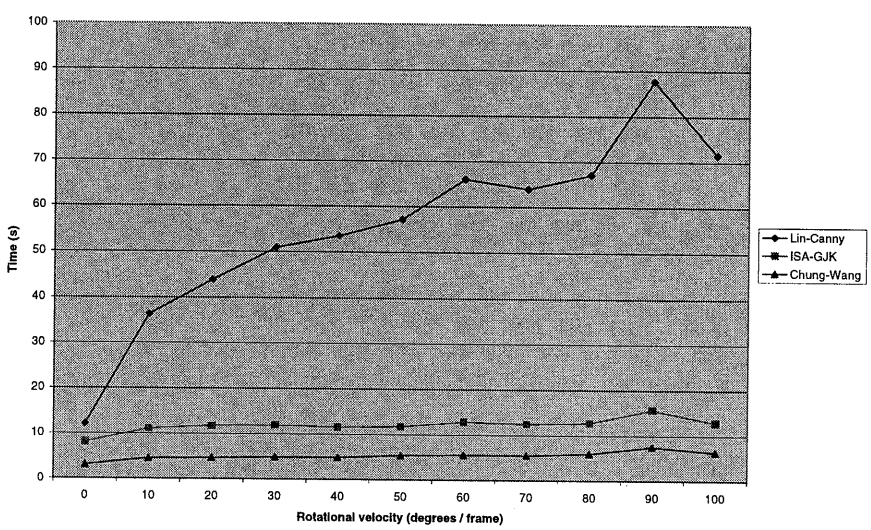


Figure 4.10: Performance under rotational-velocity variations

4.4.4 Improving Robustness

Numbers represented by a machine have finite precision. Therefore, arithmetic operations will introduce rounding errors. In this section we discuss the implications of rounding errors for the GJK algorithm, and present solutions to problems that might occur as a result of these.

Termination Condition

Let us review the termination condition $\|\mathbf{v}\| - \mu \leq \varepsilon$ of the GJK distance algorithm. We see that for large $\|\mathbf{v}\|$ the rounding error of $\|\mathbf{v}\| - \mu_k$ can be of the same magnitude as ε. This may cause termination problems. We solve this problem by terminating as soon as the relative error, rather than the absolute error, in the computed value of $\|\mathbf{v}\|$ drops below a tolerance value $\varepsilon > 0$. Thus, as termination condition we take $\|\mathbf{v}\| - \mu \leq \varepsilon\|\mathbf{v}\|$.

Moreover, for $\mathbf{v} \approx \mathbf{0}$, we see that the right member of the inequality might underflow and become zero, which in turn will result in termination problems. This problem is solved by terminating as soon as $\|\mathbf{v}\|$ drops below a tolerance ω, where ω is a small positive number.

We would like to add that our experiments have shown that for quadric objects, such as spheres and cones, the average number of iterations used for computing the distance is $O(-\log(\varepsilon))$, i.e., the average number of iterations is roughly linear in the number of accurate digits in the computed distance. For polytopes, the average number of iterations is usually less than for quadrics, regardless of the complexity of the polytopes, and is not a function of ε (for small values of ε).

Backup Procedure

The main source of GJK's numerical problems due to rounding errors is the computation of $\Delta_i(X)$. Each nontrivial $\Delta_i(X)$ is the product of a number of factors of the form $\mathbf{y}_i \cdot \mathbf{y}_k - \mathbf{y}_i \cdot \mathbf{y}_j$. If $\mathbf{y}_k$ is almost equal to $\mathbf{y}_j$ in one of these factors, i.e., X is close to being affinely dependent, then the value of this factor is close to zero, in which case the relative rounding error in the machine representation of this factor may be large due to numerical cancellation. This results in a large relative error in the computed value of $\Delta_i(X)$, causing a number of irregularities in the GJK algorithm.

One of these irregularities was addressed in the original paper [40]. Due to a large relative error, the sign of the computed value of $\Delta_i(X)$ may be incorrect. As a result of this, the subalgorithm will not be able to find a subset X that satisfies the stated criteria. The original GJK uses a backup

procedure to compute the best subset. Here, the best subset is the subset X for which all $\Delta_i(X)$-s are positive and $v(\text{aff}(X))$ is nearest to the origin.

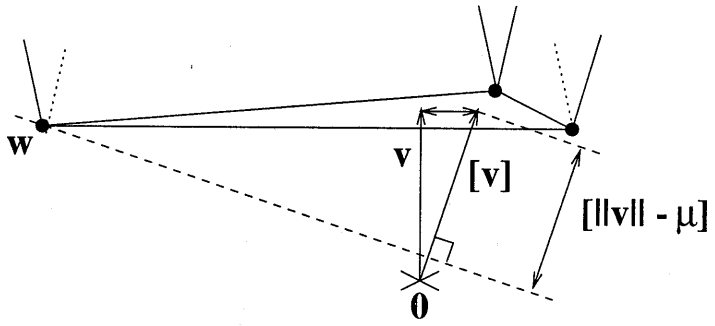
In our experiments, we observed that, in the degenerate case where the backup procedure needs to be called, the difference between the best vector returned by the backup procedure and the vector $\mathbf{v}_k$ from the previous iteration is negligible. Hence, considering the high computational cost of executing the backup procedure, we chose to leave it out and return the vector from the previous iteration, after which GJK is forced to terminate. Should the algorithm continue iterating after this event, then it will infinitely loop, since each iteration will result in the same vector being computed.

Ill-conditioned Error Bounds

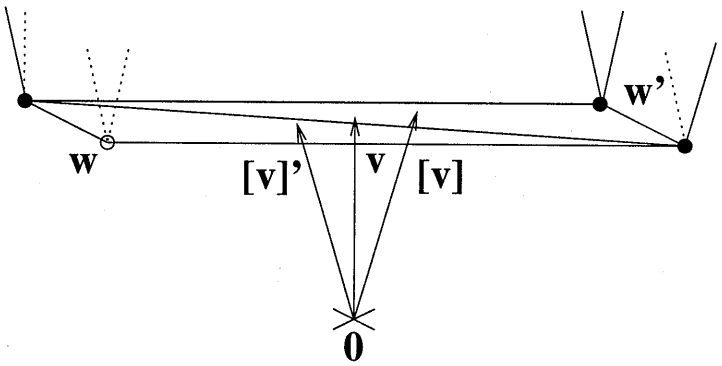
Despite these precautions, the algorithm may still encounter configurations of objects that cause it to loop infinitely, as noted by Nagle [71]. This problem may occur when two polytopes that differ a few orders of magnitude in size are in close proximity of each other. Due to the difference in size, the Minkowski sum of the objects has extremely oblong shaped facets. Let us examine a scenario in which the current simplex $\text{conv}(W)$ is an oblong shaped triangle, and $\mathbf{v} = v(A - B)$ is an internal point of the triangle.

First we note that two of the triangle's vertices lie close to each other. This may cause a large relative rounding error in the computation of $\Delta_i(X)$ for some subsets X. Hence, the computed value $[\mathbf{v}]$ of $\mathbf{v}$ might suffer from this error. Note that the subalgorithm always computes λ_i-s that are positive and add up to one. Thus, $[\mathbf{v}]$ is also an interior point of the triangle, yet located at some distance from $\mathbf{v}$. Figure 4.11(a) depicts the effect of an error in $[\mathbf{v}]$. We see that a small error in $[\mathbf{v}]$ may result in a large error in $[\|\mathbf{v}\| - \mu]$, the computed error bound of $\|\mathbf{v}\|$. The algorithm should terminate at this point since the actual $\|\mathbf{v}\| - \mu$ is zero. However, the error in $[\|\mathbf{v}\| - \mu]$ causes the algorithm to continue iterating. Since the support point $\mathbf{w}$ for $[\mathbf{v}]$ is already a vertex of the current simplex, the algorithm will find the same $[\mathbf{v}]$ in each following iteration, and thus, will never terminate.

Another problem occurs when $v(A - B)$ lies close to the diagonal of an oblong quadrilateral facet in $A - B$, as depicted in Figure 4.11(b). Again, the large error in $[\|\mathbf{v}\| - \mu]$ causes the algorithm to continue iterating. Only this time, GJK alternately returns the diagonal's opposing vertices $\mathbf{w}$ and $\mathbf{w}'$ as support points. In each iteration, one of the vertices is added to the current simplex and the other is discarded, and vice versa. The remaining two vertices of the current simplex are the vertices of the facet lying on the



(a) Same support point in each iteration



(b) Alternating support points

Figure 4.11: Two problems in the original GJK, resulting from ill-conditioned error bounds.

diagonal. We see that for the simplex containing $\mathbf{w}$, the value $[\mathbf{v}]'$ is computed, which results in the vertex $\mathbf{w}'$ being added to the current simplex. For the simplex containing $\mathbf{w}'$, the value $[\mathbf{v}]$ is computed, which again will cause $\mathbf{w}$ to be added to the current simplex.

Both degenerate cases are tackled in the following way. In each iteration, the support point $\mathbf{w}_k$ is tested whether it is a member of $W_{k-1} \cup \{\mathbf{w}_{k-1}\}$. This can only be true as a result of one of the degenerate cases. If a degenerate case is detected, then the algorithm terminates and returns $[\mathbf{v}_k]$ as the best approximation of $v(A - B)$ within the precision bounds of the machine.

We have done some extensive bench tests on random input in order to compare the robustness of our enhanced algorithm with the original GJK. The tests showed that with this extra termination condition, our algorithm terminates properly for all tested configurations of polytopes, regardless of the size of the tolerance for the relative error in the computed distance, whereas the original GJK occasionally failed to terminate on the same input.

4.5 Conclusions

The best performance for collision detection of polytopes is obtained using by incremental algorithms that exploit frame coherence. This category includes the Lin-Canny algorithm, and its variant, V-Clip, the Chung-Wang algorithm, Enhanced GJK, and our ISA-GJK. Except for Chung-Wang, source code for all these algorithms is currently publicly available.

All these algorithms have a running time per frame that is roughly constant. Lin-Canny, V-Clip, and Enhanced GJK compute a closest point pair, whereas Chung-Wang and ISA-GJK find a separating axis. Overall, the incremental separating axis algorithms have the best performance, and are therefore most suitable for interactive collision detection. As an added bonus, ISA-GJK presented here is, besides polytopes, applicable to other convex objects as well, for instance to quadric primitives, such as cones, and cylinders.

In the application of collision detection to physics-based simulations, it is necessary to have (an approximation of) a contact plane in order to compute the direction of the reaction forces to a collision. As we saw in Chapter 2, a contact plane can be approximated using a closest point pair from the frame prior to the collision frame. Obviously, we can use a closest point tracking algorithm for finding this closest point pair. However, since only the closest point pair from the frame prior to the collision frame is

needed, the following strategy, suggested in [18], is likely to be faster.

For detecting intersections, an incremental separating axis algorithm, such as ISA-GJK, is used. On detecting a collision, we back up to the previous frame and compute the closest points for this frame using a closest point algorithm, such as the GJK distance algorithm. In general, the added cost of the closest point computation is largely made up for by the better performance of the separating axis algorithm.

Chapter 5

Spatial Data Structures

"I feel like a million tonight, but one at a time."
Mae West

So far, we have discussed intersection tests for a variety of shape types. For collision detection of models composed of thousands of objects, the number of intersection tests that need to be performed can become quite large. In this chapter, we will discuss a number of spatial data structures that can be used to avoid doing a lot of intersection tests for complex models.

Spatial data structures are used in two ways. Firstly, they can be used to reduce the number of intersection tests among a large number of freely moving objects in a scene. Secondly, they are used to reduce the number of pairwise primitive intersection tests in intersection testing between two complex models composed of a large number of primitives.

An important concept in this context is **geometric coherence**, as defined in Chapter 2. Geometric coherence is important, since it allows us to quickly reject pairs of objects from intersection testing based on the region of space that they occupy. Spatial data structures, such as voxel grids, hierarchical space partitioning structures, and bounding volume hierarchies can be used for 'capturing' geometric coherence. Basically, the data structures that are used for this purpose fall into two categories: space partitioning, and model partitioning. In the following sections, we will discuss the merits and drawbacks of data structures for each category.

5.1 Space Partitioning

A space partitioning is a subdivision of space into convex regions, called cells. Each cell in the partitioning maintains a list of references to objects

that are (partially) contained in the cell. Using such a structure, a lot of object pairs in the model can be quickly rejected from intersection testing, since we only need to test the pairs of objects that share a cell. A very straightforward space partitioning is a voxel grid.

5.1.1 Voxel Grids

A voxel grid is a space partitioning into uniform rectangular cells. A voxel grid is represented by an axis-aligned box enclosing all objects in the model, and a three-dimensional array of cells of size $N_x \times N_y \times N_z$, where N_x, N_y, and N_z are the number of subdivisions along the respective axes. Other than the position and size of the box, no further geometric data need to be maintained in the grid structure.

Let $(\beta_x, \beta_y, \beta_z)^T$ be a vertex of the box and η_x, η_y, and η_z, the extents along the respective axes, such that the box is the Cartesian product of the intervals $[\beta_k, \beta_k + \eta_k)$ for $k = x, y, z$. Then, the cell indexed by (i_x, i_y, i_z) is the box defined by the product of intervals

$$[\beta_k + i_k \frac{\eta_k}{N_k}, \beta_k + (i_k + 1)\frac{\eta_k}{N_k}) \quad \text{for } 0 \leq i_k < N_k \text{ and } k = x, y, z.$$

Each cell maintains a list of objects that intersect with the cell.

Assume we have a collection of objects of more or less the same size which are represented by primitive shapes, such as boxes or spheres, for instance used as bounding volumes, and assume that the set has a high degree of geometric coherence and is distributed evenly over the space. Then, for a grid that has a number of cells that is linear in the number objects, adding or moving an object can be done in constant time [16, 58]. This is useful in an environment with lots of similar moving objects such as described in [107].

Voxel grids have been shown to be useful also in real-time intersection testing between complex rigid objects [37, 38]. Here, each object's primitives are maintained in a grid that is aligned to the local coordinate system of the object. The algorithm finds all pairs of overlapping nonempty cells of a pair of relatively oriented grids, and tests for intersections between primitives for these pairs of cells.

The benefits of using a voxel grid are low storage usage, and fast cell access. However, the biggest drawback of voxel grids, which is common to all bucketing techniques, is the fact that performance greatly depends on the density of objects in the space being uniform. If the objects in an environment are clustered, which is commonly the case in many applications, then a few cells contain most of the objects, whereas the majority

of cells are empty. Hence, the grid is not very useful in rejecting pairs of objects for intersection testing.

Another drawback is the fact that the best resolution of a voxel grid for a given model is hard to estimate. If there are too few cells in the grid, then the cells will be crowded and thus a lot of intersection tests will be performed among objects in each cell. If there are too many cells, then the grid takes up a lot of storage, and many cells will maintain references to identical objects. This results in a lot of intersection tests being repeated over and over again, since often a number of cells will maintain the same pair of objects.

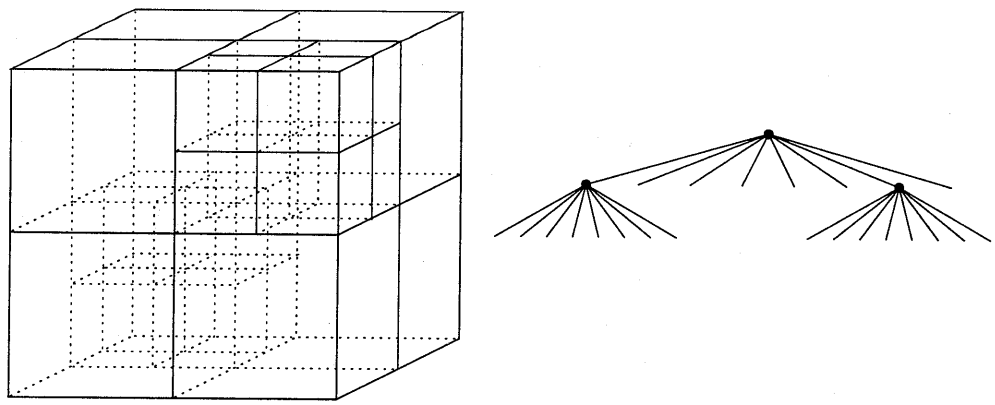
Of course, we may keep a record of all the pairs of objects that are tested for intersection in order to avoid repeating intersection tests, similar to mailbox methods used in ray tracing [41]. However, the overhead of maintaining these data results in added computational and storage costs. Also, inserting an object that overlap many cells is expensive, since a reference to this object needs to be inserted in each cell.

Voxel grids work best for geometrically coherent sets of primitives of uniform density and size. However, in practice, few models satisfy these conditions. Hence, adaptive partitioning schemes, using recursive space partitioning, often yield better results. In the following sections, we will discuss a number of hierarchical space partitioning methods that are better suited for dealing with clusters of objects.

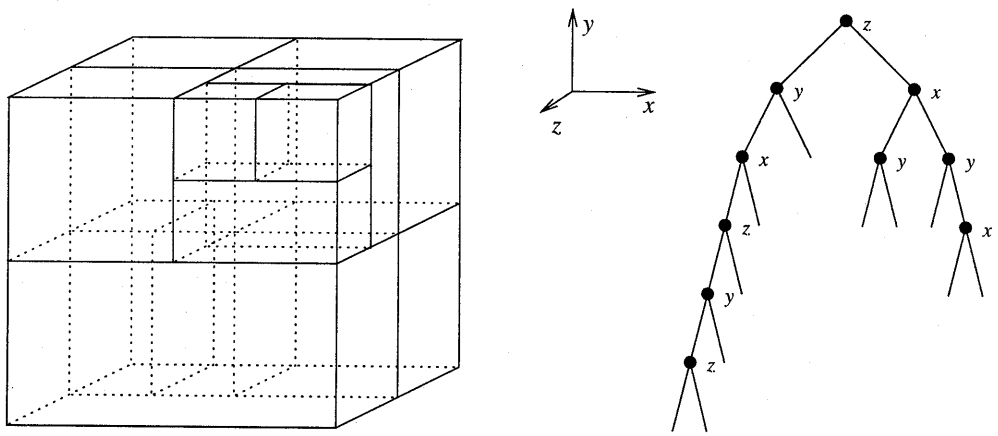
5.1.2 Octrees and k-d Trees

Octrees and k-d trees are hierarchical structures for partitioning space into rectangular cells. Each node in the tree corresponds to a rectangular region of the space. The root node corresponds to the whole space represented by a rectangular box enclosing the complete model. Each internal node in the tree represents a subdivision of its corresponding region into smaller regions which correspond to the children of the node. The regions of the leaf nodes are the cells in which the objects are maintained.

Octrees and k-d trees differ in the way the regions are subdivided. Each internal node in an octree divides its corresponding region into eight octants by splitting the region along the three coordinate axes. In a k-d tree, the region of an internal node is subdivided into two regions by splitting the region along an arbitrary coordinate axis. See Figure 5.1 for a visual representation of the two structures. It can be seen that the best k-d tree often has fewer cells than the best octree for the same configuration of objects.

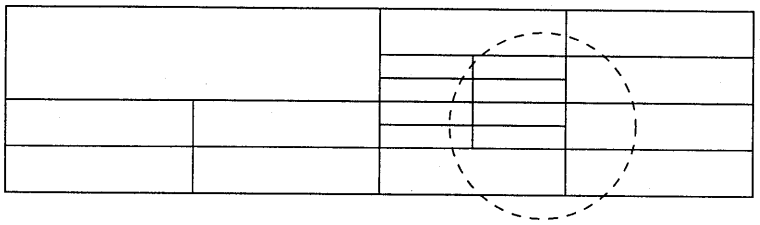


(a) Octree

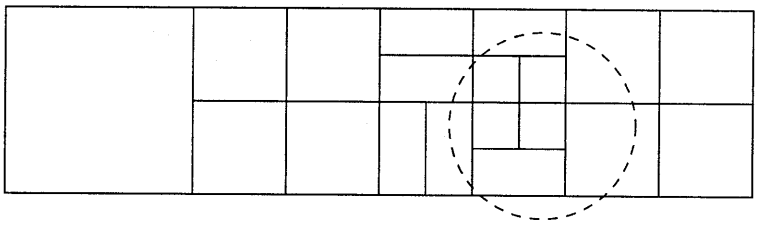


(b) *k*-d tree

Figure 5.1: Two hierarchical structures for partitioning space into rectangular cells



(a) Octree



(b) k-d tree

Figure 5.2: A query object (dashed circle) can overlap less fat cells than thin cells.

In the standard representation, a region is split into subregions of equal size. However, octree or k-d tree variants, in which regions are split at arbitrary points along the axis, are possible as well. These tree variants require more storage, since the positions of the partitioning planes need to be stored in the internal nodes, but the added freedom in choosing the partitioning results in a structure that is more adaptive to the configuration of objects.

It can be seen that in a standard octree each node's region has the same aspect ratio as the root's region. This property often results in poor partitionings if the root region is oblong. Often, partitioning a space into fat cells (aspect ratio $\approx$ 1) results in better performance for intersection queries than a partitioning into thin cells, since a given query object can overlap more thin cells than fat cells of the same volume, as depicted in Figure 5.2. Hence, in the cases where the root's region is oblong, it is better to use k-d trees, rather than octrees, since in a k-d tree a cell's aspect ratio need not be equal to the root region's aspect ratio. Of course, for octrees the root region can be taken to be a cube enclosing the objects in the model, however, this may result in a larger number of cells of which some are al-

ways empty. A more general definition of fatness and its significance to space partitioning is presented in [25].

The benefit of using a recursive space partitioning is the fact it is adaptive to local densities in a model. In regions where a lot of objects are clustered the space is partitioned into many small cells, whereas in regions where there are hardly any objects the cells can be large. In environments in which the local densities of objects change over time, for instance, as in simulations of chemical phenomena [97], dynamic hierarchical space partitioning structures may be applied. If a cell becomes crowded it can be subdivided into smaller cells containing less objects. Conversely, if the children of a node, which are all leaves, contain few objects, the corresponding cells can be united and the internal node is transformed into a leaf.

Octrees and k-d trees can be used for reducing the number of pairwise intersection tests among objects in a model, as well as for reducing the number of intersection tests between primitives from a pair of complex models. An example of a k-d-tree-type data structure that is used in the latter application is the *BoxTree* by Zachmann and Felger [106]. A BoxTree is constructed a priori for each complex model in its local coordinate system. It is a static structure, and is therefore applicable to rigid, i.e., non-deformable, models only. Examples of uses of octrees for deformable models are described in [68, 88]. In both methods, a single octree, maintaining primitives for different objects in the scene, is constructed anew for each frame. In Section 5.2 we will present a new scheme for detecting collisions among deformable models, which is based on axis-aligned bounding box trees.

Octrees and k-d trees are popular space partitioning structures, since they require relatively little storage and are adaptive to differences in local densities in a model. Next, we meet the most general of all recursive space partitioning structures, the binary space partitioning tree.

5.1.3 Binary Space Partitioning Trees

A **binary space partitioning** (BSP) tree is a hierarchical structure for partitioning space recursively into convex cells. Each internal node in the tree divides the convex region associated with the node into two regions by means of a freely oriented partitioning plane. Phrased differently, a BSP tree is a variable split k-d tree, but without the restriction that the partitioning planes are oriented orthogonal to the coordinate axes. A variant of the BSP tree in which the orientations of the partitioning planes are chosen

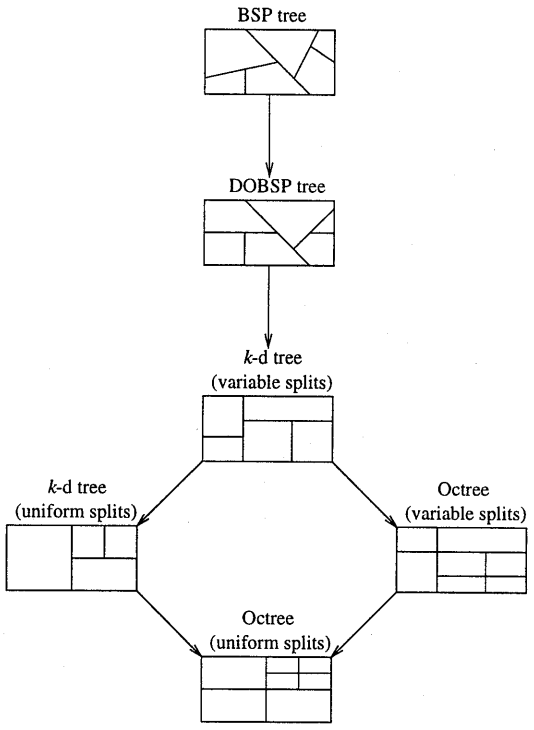


Figure 5.3: A taxonomy of recursive hierarchical space partitioning structures. The arrows in the diagram denote the relation "is a generalization of".

from a finite set of orientations, similar to DOPs, may be useful in some applications. We refer to this type of BSP tree as **discrete-orientation BSP** (DOBSP) tree. Representing cell structures using DOBSP trees requires less storage than using general BSP trees, since in a DOBSP tree the partitioning plane orientations are described by an index number rather than a 3D vector. Figure 5.3 shows a taxonomy of recursive hierarchical space partitioning structures.

BSP trees can be used to represent cell structures in which objects are maintained. However, a more interesting application of BSP trees is found in representing non-convex polyhedra, as an alternative to boundary representations. As we saw in Chapter 3, a non-convex polyhedron can be represented as the union of a subset of cells in a binary space partitioning. For this purpose, the leaves of the BSP tree are labeled *in* or *out*, depending on whether the corresponding cell lies inside or outside the polyhedron [95], as depicted in Figure 5.4.

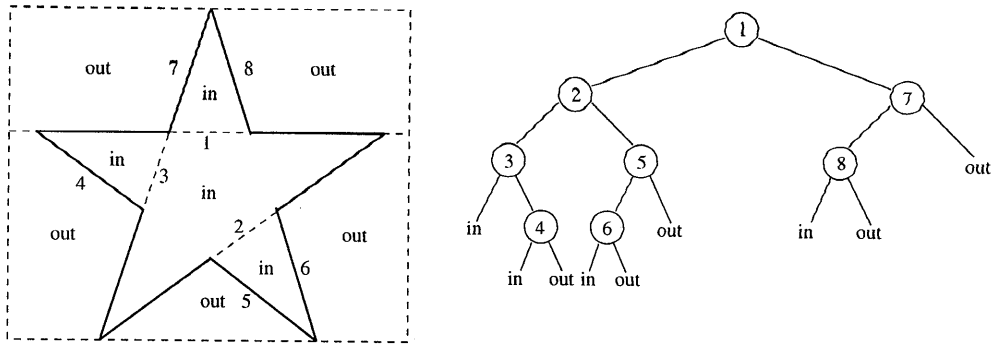


Figure 5.4: A polygon and its BSP tree representation

In [73], an algorithm is presented for performing boolean operations on a pair of polyhedra represented by BSP trees. The result of a boolean operation, which is also represented by a BSP tree, is obtained by merging the two BSP trees. A merge of a pair of BSP trees involves splitting one of the trees according to the root partitioning plane of the other tree, and recursively merging the split trees with the corresponding children of the latter tree. In order to allow efficient splits, each BSP tree maintains, besides a partitioning plane, also a convex polygon in each internal node. The polygon represents the intersection of the partitioning plane and the convex region associated with the node, and is explicitly represented by a list of vertices.

This merge operation can be used for efficiently computing the intersection of a pair of non-convex polyhedra. The algorithm can also be applied to testing the intersection between a pair of polyhedra, with some adaptions for speed to allow early exit in case an intersection is found [74]. However, note that these adaptions will only marginally improve performance for intersection testing, since it seems that the BSP tree split, the most expensive operation in the intersection computation, cannot be removed or simplified for the intersection detection problem.

BSP tree representations of polyhedra are expensive with respect to the storage requirements, especially the ones used for fast tree merges, which maintain a convex polygon in the internal nodes. In [79], Paterson and Yao show how to construct a BSP tree of $O(n^2)$ nodes, representing a polyhedron of n facets. Furthermore, they prove that this bound is optimal, i.e., polyhedra exist for which the smallest BSP tree representation still has $\Omega(n^2)$ nodes. Due to their large storage usage and high computational cost, we consider it therefore unlikely that collision detection algo-

rithms based on BSP tree representations will outperform algorithms that use boundary representations.

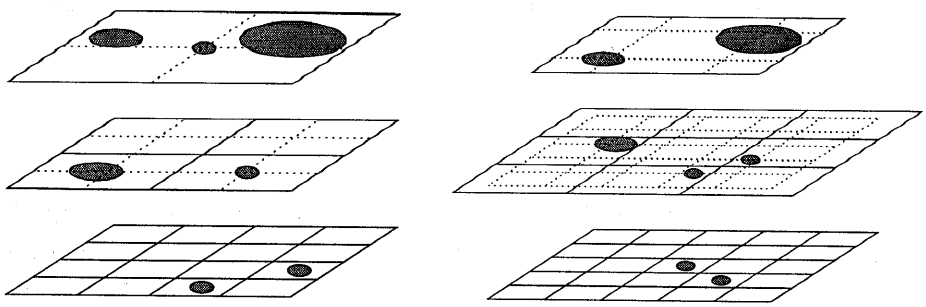
5.1.4 Discussion

A drawback of space partitioning methods is the fact that objects that straddle cell boundaries are maintained in multiple cells. This may lead to structures with sizes that exceed the number of stored objects by orders of magnitude. Maintaining multiple references to the same object results in either a lot of intersection tests being repeated for the same pair of objects, or additional overhead for keeping record of the pairs of objects that have been tested for intersection.

Hence, it seems better to store objects in search structures in such a way that each object is referred to only once. For hierarchical space partitioning structures, this can be achieved by maintaining objects in all the nodes rather than in the leaves only. Each object is maintained in the node corresponding to the smallest region that encloses the object. The **interval tree** is an example of such a data structure for one-dimensional space, and is used for efficiently reporting all intervals in a set that intersect a query interval [80].

However, when we allow objects to be stored in all nodes rather than leaf nodes only, intersection testing among objects is a little more complicated, since intersecting objects may be maintained at different levels in the tree. For finding all intersecting objects, we fall back on range queries in which we use the objects' axis-aligned bounding boxes as query windows. The computational cost of a range query depends on the number of objects maintained in nodes that are visited during the query. A node is visited if the corresponding region overlaps with the query box.

A problem that occurs when objects are maintained in this way in recursive space partitioning trees (octrees, k-d trees, BSP trees), is that small objects may be stored at all levels in the tree, i.e., there is no upper bound for the extent of the region in relation to the size of the object stored in the corresponding node. Figure 5.5(a) illustrates this property for a quadtree. Hence, the upper levels of the tree may contain an unacceptable number of objects. Since the nodes at the upper levels are visited more often than the nodes close to the leaves, the range queries are slowed down considerably by a lot of small objects at upper levels. We say 'slowed down', since these objects have a low probability of intersecting with the query object, due to their size. Therefore, most of the intersection tests on small objects at upper levels are wasted.



(a) Quadtree (b) Fieldtree

Figure 5.5: Two spatial data structures used in GIS. In both structures the objects are stored in the node corresponding to the smallest region enclosing the object. Notice that in the quadtree small objects are stored at all levels, whereas in the fieldtree small objects are stored close to the leaves only.

A data structure known from geographic information systems (GIS) literature, referred to as **fieldtree**, can be used to solve the problem of having small objects stored in nodes at upper levels [34]. In a fieldtree, a node may have up to nine children, and a child node may have multiple parents, hence, a fieldtree is actually a directed acyclic graph (DAG). The extent of a child's region is half the extents of its parents' regions as in a quadtree, however, the different levels of the tree are shifted with respect to their parent nodes. The shift of region boundaries allows objects to be stored in a node for which the size of the region is approximately the size of the objects. Thus, large objects are stored close to the root, whereas smaller objects are stored close to the leaves. Figure 5.5(b) shows how objects are stored in a fieldtree.

A generalization of the fieldtree to three-dimensional space is possible, and may be useful as an alternative to the voxel grid. However, we expect such a structure to be less useful as a dynamic data structure, since the time complexity of adding and deleting nodes in the tree is quite high due to the fact that the children may have multiple parents.

To conclude this survey of space partitioning structures, we present an overview of structures that are, or may be, used for speeding up collision detection. Depending on the type of intersection problem, we recognize the following solutions using space partitioning techniques:

- Finding all intersecting pairs among a set of moving objects.
  - Evenly distributed objects of more or less equal size.

    Voxel grid [107, 58] Merits: fast access and update times, low storage cost. Drawbacks: bad performance due to multiple references to objects that straddle cell boundaries.
  - Changes in local densities of objects over time.

    Octree, k-d tree, (DO)BSP tree Merits: adaptive to differences in local densities of objects, low storage cost (octree, k-d tree, DOBSP tree). Drawbacks: bad performance when objects straddle cell boundaries: either maintain objects in the leaves (cells) only and keep multiple references to objects, or maintain objects in all nodes, possibly resulting in small objects being stored close to the root, and thus inefficient range queries.
  - Wide variations in object sizes.

    3D fieldtree Merits: fast query times due to single references to object, and size-sensitive storage of objects. Drawbacks: complex structure and algorithms, fairly high storage cost.
- Finding intersecting primitives between a pair of complex models.
  - Rigid models

    Voxel grid [37, 38] Merits: fast access, low storage cost. Drawbacks: bad performance due to multiple references to objects that straddle cell boundaries.

    k-d tree [106] Merits: adaptive, low storage cost. Drawbacks: bad performance when objects straddle cell boundaries.
  - Deformable models

    Octree [68, 88] Merits: fast construction times, low storage cost. Drawbacks: bad performance when many objects straddle cell boundaries.

In the following section, we discuss spatial data structures in which the set of objects in a model is partitioned rather than the space in which the objects are placed.

5.2 Model Partitioning

Model partitioning is often a better choice than space partitioning, since model partitioning structures do not suffer from the problem of having multiple references to the same object. The basic strategy is to subdivide a set of objects into geometric coherent subsets, and compute a tight-fitting bounding volume for each subset of objects, such that in intersection tests, subsets of objects can be quickly rejected from intersection testing depending on whether their bounding volumes overlap. In the following section we discuss a number of commonly used bounding volume types.

5.2.1 Bounding Volumes

A bounding volume of a model is a primitive shape that encloses the model and has the following desired properties:

1. A bounding volume should fit the model as tightly as possible in order to have a low probability of a given query object intersecting the volume but not the model.

2. Overlap tests between bounding volumes should be computationally cheap. In particular, cheaper than testing the enclosed models for intersection.

3. A bounding volume should be representable using a small amount of storage, thus, the additional storage space used by the spatial data structure is small in comparison to the storage used by the model.

4. The cost of computing a bounding volume for a given model should be low. This property is relevant only if the volume is recomputed regularly.

Examples of bounding volume types are: spheres, axis-aligned bounding boxes (AABBs), discrete-orientation polytopes (DOPs), and oriented bounding boxes (OBBs). An OBB's orientation is represented by a 3×3 matrix, which defines a local basis with respect to the model's local coordinate system. Although for rectangular OBBs, a quaternion representation of the orientation uses less storage space (4 vs. 9 scalars), a matrix representation is more efficient in overlap tests [46].

Of the mentioned volume types, only spheres and OBBs are independent of the model's orientation. The extents of AABBs and DOPs are either recomputed whenever the enclosed model is rotated, or set large enough,

such that the model can be enclosed in all possible orientations. Computing the smallest AABBs and DOPs for sets of vertices can be done reasonably fast. We simply compute the projections of the vertices on the given axes and maintain the minimum and maximum value for each axis. OBBs and spheres need to be recomputed only if the enclosed objects are deformed or scaled, although we will present an application of OBBs further on, in which the boxes are independent of nonuniform scalings of the enclosed models.

For OBBs, finding the best orientation is quite complex, however, we can get reasonably tight-fit boxes by applying heuristics. In [103], Wu presents a method that uses multivariate analysis to find the principal axes of the distribution of the vertices. The principal axes are mutually orthogonal, and can therefore serve as a local basis for rectangular boxes. This method results in tight-fit boxes for many models, however, for some cases, such as the vertices of a cube, this method returns orientations that are quite bad. Gottschalk proposes in [46] a heuristic that uses axes based on the weighted spread of sample points on the surface of the convex hull of the set of vertices in order to get a tight fit for a wider range of models. Evidently, this method is much more expensive, since a convex hull computation requires $\theta(n \log n)$ time for n vertices.

Computing the smallest enclosing sphere can be done in expected linear time using a randomized algorithm [101], however the constant factor of this algorithm is quite large. We can get a reasonable approximation of the smallest sphere using the following heuristic presented in [103]. First, we compute a local basis of principal axes as used for OBB computations. For each axis i, we find the extreme vertices, i.e., the vertices $\mathbf{p}^i_{\min}$, $\mathbf{p}^i_{\max}$, for which the projection on i is the minimum, respectively maximum, of all vertices. We construct a tentative sphere with diameter the line segment connecting the vertices $\mathbf{p}^i_{\min}\mathbf{p}^i_{\max}$ for the axis i for which $d(\mathbf{p}^i_{\min}, \mathbf{p}^i_{\max})$ is the largest. Next, for each vertex that falls outside of the tentative sphere, we expand the sphere such that it contains the vertex and the former sphere.

Table 5.1 shows an overview of the characteristics of the mentioned bounding volume types. We see that a tighter fit is acquired at a higher cost of storage space and processing time for overlap testing. The choice of bounding volumes for a particular application depends on the shapes and complexities of the enclosed objects, as well as the densities of the objects in the space.

Let us discuss a real-life example of the use of bounding volumes on the basis of the average time formula presented in Chapter 2. Recall that

Volume	Fit	Cost overlap test (ops)	Storage (scalars)
Sphere	poor	11	4
AABB	poor	9	6
k-DOP	fair	$3k$	$2k$
OBB	good	200 (cf. [46])	15

Table 5.1: A comparison of a number of bounding volume types. Here, *Cost* denotes the cost of testing a pair of volumes for overlap in number of primitive arithmetic operations, and does not include the cost of recomputing the volume after a rotation of the enclosed model.

the average time of a sequence of intersection tests can be expressed as

$$T_{\text{avg}} = \sum_{i=1}^{n} P[f_1 \cdots f_{i-1}]C_i,$$

where f_i represent the event that test i fails, and C_i, the average time necessary for performing test i. A bounding volume overlap test fails iff the volumes overlap.

For this example, we use a model of a torus composed of a large number of triangles. The torus has a major radius of 10 and a minor radius of 2. The test is performed by placing a pair of tori randomly in space, and testing them for intersection. The tori have arbitrary orientations. The space size in which the tori are placed is determined by a cube in which the centers of the tori are randomly positioned. By scaling the cube the probability of an intersection can be tuned.

We first examine the use of a single bounding volume overlap test. The average time formula for such a test is given by

$$T_{\text{avg}} = C_{\text{volume}} + P[f_{\text{volume}}]C_{\text{torus}}.$$

The cost of testing a pair of tori for intersection, denoted here by C_{torus}, depends on the number of triangles, but should at its best be in the order of 10^4 operations for a torus composed of a hundred triangles, so for the sake of our discussion let us assume $C_{\text{torus}} = 10000$ ops. Table 5.2 shows the average times over 100,000 runs for a bounding sphere and an OBB. The smallest sphere enclosing the torus has a radius of 12. The smallest OBB has a size of $24 \times 24 \times 4$. The cost of overlap testing can be found in Table 5.1 for both volume types. We see that for both bounding volume types the use of the overlap test is justified, since the average time when using the volume is smaller than C_{torus} for the tested space sizes. Moreover,

Bounding sphere		
Cube size	$P[f_{\text{sphere}}]$ (%)	T_{avg} (ops)
100	4.6	471
20	98.7	9881
OBB		
Cube size	$P[f_{\text{OBB}}]$ (%)	T_{avg} (ops)
100	4.1	610
20	86.9	8890

Table 5.2: Average performances of an intersection test on a pair of tori, using a single bounding volume overlap test. The value of *Cube size* is the length of the sides of the cube in which the centers of the tori are placed.

we see that the bounding spheres perform better than the OBBs when the density is low, whereas OBBs perform better for high densities.

Often, it makes sense to use a combination of bounding volume types. A cheap loose-fit volume type yields quick rejections of pairs of objects that are at some distance of each other, and a tighter more expensive volume type is used for rejecting closer configurations of objects. The average time formula for a test with two bounding volume types is given by

$$T_{\text{avg}} = C_{\text{volume}_1} + P[f_{\text{volume}_1}]C_{\text{volume}_2} + P[f_{\text{volume}_1} f_{\text{volume}_2}]C_{\text{torus}}.$$

We examine this idea using OBBs as the tight-fitting volumes in the second bounding volume test. For the choice of the first volume we have several options.

Our first option is to use a dynamic AABB enclosing the OBB. A dynamic AABB is recomputed for each placement of the torus. Although the smallest AABB enclosing the torus is usually smaller than the smallest AABB enclosing the OBB, we opt for the latter construction, since computing the smallest AABB of an OBB is much cheaper than computing the smallest AABB of the set of triangles that represents the torus. The smallest AABB enclosing an OBB can be computed using 24 operations. For this purpose, we compute the projections of the OBB onto the coordinate axes, which takes 8 operations for each axis, as we saw in Chapter 3. Hence, the total cost for performing an overlap test for a pair of AABBs is two times 24 operations for the AABB computations plus 9 operations for the actual overlap test, which makes a total of 57 operations. See Table 5.3 for the results of this experiment. Notice that this combined approach performs better than the bounding sphere only for the high-density case, and better

Dynamic AABB & OBB			
Space size	$P[f_{\text{AABB}}]$ (%)	$P[f_{\text{AABB}} f_{\text{OBB}}]$ (%)	T_{avg} (ops)
100	9.1	4.1	485
20	99.0	86.9	8945
Fixed-size AABB & OBB			
Space size	$P[f_{\text{AABB}}]$ (%)	$P[f_{\text{AABB}} f_{\text{OBB}}]$ (%)	T_{avg} (ops)
100	7.8	3.9	415
20	100	86.9	8899
Bounding sphere & OBB			
Space size	$P[f_{\text{sphere}}]$ (%)	$P[f_{\text{sphere}} f_{\text{OBB}}]$ (%)	T_{avg} (ops)
100	4.6	3.4	360
20	98.7	86.5	8858

Table 5.3: Average performances when combining two bounding volume types

than the single OBB only for the low-density case.

Due to the recomputations, the cost of overlap testing for dynamic AABBs is high compared to the cost for fixed-size AABBs, i.e., AABBs that are set at a fixed-size that is large enough to enclose the model in each configuration. An overlap test for fixed-size AABBs takes only 9 operations. Moreover, since the cost of computing the AABB is not an issue for fixed-size AABBs, we can use the smallest fixed-size AABB of the model rather than the model's OBB. For the torus we use a fixed-size AABB with sides of length 24. As can be seen in Table 5.3, the fixed-size AABB performs better than the dynamic AABB for both space sizes, even though the overlap test for fixed-size AABBs is wasted in the high-density case ($P[f_{\text{AABB}}] = 100\%$, since two cubes of size 24 placed such that their centers lie in a cube of size 20 always overlap).

Finally, we tested a combination of a bounding sphere and an OBB. Table 5.3 shows that this is the best combination for our torus model. The sphere-OBB test shows the best performances for both space sizes of all the tests performed in this example. Also, notice that there are configurations of tori for which the bounding spheres do not overlap but the OBBs do overlap. This is shown by the fact that $P[f_{\text{sphere}} f_{\text{OBB}}] < P[f_{\text{OBB}}]$.

Generally speaking, bounding spheres yield in most cases better performance than fixed-size AABBs for intersection testing of objects that have three rotational degrees of freedom, since in this case, the smallest bounding sphere is enclosed by the smallest fixed-size AABB, and thus,

is tighter than the AABB, whereas the cost of a sphere intersection test is only slightly higher than the cost of an AABB intersection test.

5.2.2 Collections of Bounding Volumes

Although bounding volume tests are relatively cheap, testing all $\binom{n}{2}$ pairs in a collection of n bounding volumes still gives rise to a lot of computations. In cases where there is some geometric coherence in the configuration of objects, only a small number of all pairs of bounding volumes are overlapping. Hence, in these cases, an output-sensitive algorithm for finding all overlapping pairs of volumes should perform better than the naive $O(n^2)$ test-all-pairs algorithm.

An output-sensitive algorithms for n AABBs has been presented by Six and Wood in [87]. Their algorithms has a time complexity of $O(n \log^2 n + k)$, where k is the number of overlapping pairs of boxes, and requires $O(n \log^2 n)$ space. The algorithm applies a space-sweeping technique, i.e., a plane orthogonal to a given coordinate axis is swept from $-\infty$ to ∞ iterating over the coordinates corresponding to the sides of the boxes. As the plane is swept, information pertaining to the boxes that are cut by the plane is maintained data structures. The used data structures, a segment tree and a range tree [80], allow fast insertion, deletion and query operations, such that a sub-quadratic time bound can be attained. A similar technique is applied in [54] for detecting an intersection among n spheres.

Algorithms that apply space-sweeping essentially have a worst-case lower time bound of $\Omega(n \log n)$ since it is necessary to sort the input with respect to a given coordinate axis. However, if frame coherence is high, the sorted sequence of box coordinates from a previous frame is likely to be nearly sorted in the current frame, in which case sorting will take only linear time using *insertion sort* [83]. Baraff exploits this idea in an incremental algorithm for maintaining the set of overlapping AABBs during an animation [4].

In his approach, the endpoints of the intervals of projection of each box onto the three coordinate axes, described by $[b_i^x, e_i^x]$, $[b_i^y, e_i^y]$ and $[b_i^z, e_i^z]$ for the ith box, are maintained as three sorted sequences corresponding to the three coordinate axes. Each endpoint maintains, besides its coordinate, also a reference to its box, and whether it is a lower or upper endpoint of the interval. In each frame, the coordinates of the endpoints are updated, and the three sequences are sorted using insertion sort.

Basically, insertion sort is performed as follows. Assume that the sequence is sorted up to a certain element. This element is found to be less

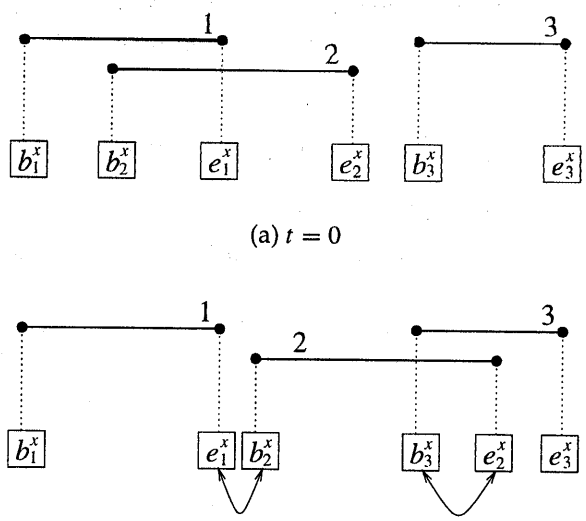


Figure 5.6: Sorting a sequence of endpoints. Box 2 moves and causes two swaps in the sequence of x-coordinates. The swap of b_2^x and e_1^x indicates that the intervals of Box 1 and 2 onto the x-axis cease to overlap. The swap of e_2^x and b_3^x indicates that the intervals of Box 2 and 3 begin to overlap.

than its predecessor, and has to be inserted in the segment of the sequence from the head up to the current position. This insertion is performed by swapping the element with its predecessor until an element is reached that is less than the current element. This process continues until we reach the end of the sequence.

When during a sort a lower endpoint of one box and an upper endpoint of another box are swapped, the intervals of the two boxes corresponding to the coordinate axis will either start or cease to overlap. Note that insertion sort performs swaps of adjacent elements only, thus the only intervals that may be affected by the swap are the ones that correspond to the swapped endpoints. Figure 5.6 illustrates how changes in overlap status of box intervals can be detected. When a pair of intervals on a coordinate axis change from non-overlapping to overlapping, the corresponding pair of boxes are tested for overlap with respect to the other two coordinate axes. If the boxes overlap, then the pair is inserted in a set of overlapping pairs of boxes. When a pair of intervals cease to overlap, the corresponding pair of boxes, are removed from the set of overlapping pairs of boxes

in case the boxes were previously overlapping.

In order to keep the times of insertions and deletions of pairs of boxes small, the set of overlapping box pairs is best implemented using a balanced binary-search tree (AVL tree, red-black tree) or a hash table [64]. Each swap can be performed in constant time, except for the swaps that result in an insertion or deletion of a box pair. Insertions and deletions of box pairs take $O(\log k)$ time for a set of k box pairs represented by a balanced binary search tree. Since a nearly sorted sequence can be sorted in linear time, we find that when frame coherence is high, the cost of updating the set of overlapping box pairs is $O(n + c \log k)$, where c is the total number of box pairs whose overlap status changed with respect to the previous frame.

Often, only a few of the objects in a scene are moving at a given time. In order to exploit this, we present an adaption of Baraff's algorithm that allows update of the set of overlapping box pairs in time linear in number of *moving* boxes. Instead of performing an insertion sort on the sequences of endpoints once for all endpoints, we propose an incremental approach in which insertions are performed immediately for each displaced endpoint.

Each time an endpoint is assigned a new position, the endpoint is immediately inserted at the correct position in the sequence. Contrary to insertion sort, where insertion are only done downward, we allow an insertion to be performed upward as well as downward. We are able to do this since both the sequence segments below and above the displaced endpoint are always sorted, whereas with insertion sort only the segment below the current endpoint is sorted. In this way, the update time per frame is expected to be $O(m + c \log k)$ when frame coherence is high, where m is the number of moving objects.

In order to keep the overhead involved in updating the endpoints low, we implement the sequences using doubly-linked lists. Each endpoint maintains pointers to its successor and its predecessor in the list. In this way, each object can maintain a direct reference to the endpoints of its AABB, without the necessity of updating the references whenever the endpoints' positions in the sequence are changed. Figure 5.7 illustrates this implementation.

5.2.3 Bounding Volume Hierarchies

We can 'capture' geometric coherence in a model by means of a bounding volume hierarchy. A bounding volume hierarchy is a tree structure in which primitives are stored in the leaves. Each node maintains a bounding

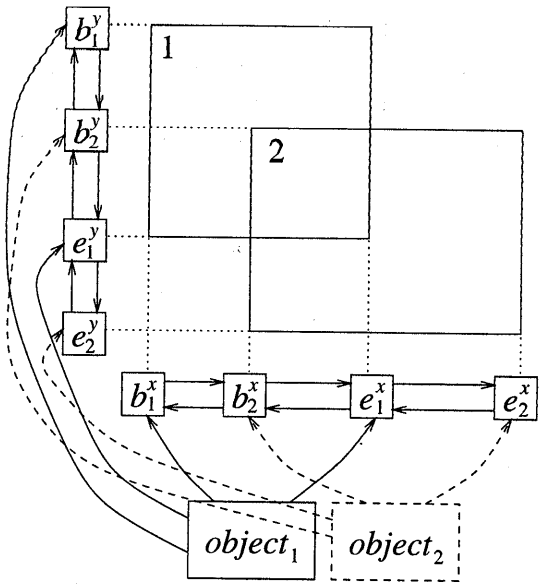


Figure 5.7: The incremental endpoint sort algorithm. For each coordinate axis, the endpoints of each object's AABB are maintained in sorted doubly-linked lists.

volume of the subset of primitives represented by the node. The bounding volumes of the children of a node may, and often do, overlap. Examples of bounding volume hierarchies are sphere trees [78, 56], oriented bounding box (OBB) trees [46], and discrete-orientation polytope (DOP) trees [59, 105].

The benefits of using bounding volume hierarchies are the fast query times for intersection testing, and the linear space requirements with respect to the number of objects in the model. The major drawback is the high cost of constructing a bounding volume hierarchy, and maintaining a hierarchy under model changes. Therefore, bounding volume hierarchies are generally used only for complex rigid models, for which construction is performed only once as a preprocessing step, although application of bounding volume hierarchies to collision detection among freely moving objects has been considered [99]. Further on, we will present a new application of AABB trees to collision detection between complex models undergoing deformation.

An intersection test between two models represented by bounding volume hierarchies is performed by recursively testing pairs of nodes. The intersection test handles the following cases:

1. If the bounding volumes of the nodes do not intersect then false is returned.

2. If both nodes are leaves then the primitives are tested for intersection and the result of the test is returned.

3. If one of the nodes is a leaf and the other an internal node, then the leaf node is tested for intersection with each of the children of the internal node.

4. If both nodes are internal nodes then the node with smaller volume is tested for intersection with the children of the node with the larger volume.

The latter heuristic choice of first unfolding the node with the largest volume results in the largest reduction of total volume size in the following bounding volume tests, thus the lowest probability of following bounding volume tests returning an intersection.

Note that we do not perform volume-primitive intersection tests. Volume-primitive tests are often as expensive as primitive-primitive tests, and have a high probability of failure, i.e., the chance of a volume-primitive test returning false is rather small. Hence, adding volume-primitive tests as a pre-step in case 2 of the recursive intersection test is likely to worsen the performance, rather than improve it.

The total cost of testing a pair of models represented by bounding volume hierarchies is expressed in the following cost function [100, 46]:

$$T_{total} = N_b * C_b + N_p * C_p,$$

where

T_{total}	is the total cost of testing a pair of models,
N_b	is the number of bounding volume pairs tested for intersection,
C_b	is the cost of testing a pair of bounding volumes for intersection,
N_p	is the number of primitive pairs tested for intersection, and
C_p	is the cost of testing a pair of primitives for intersection.

The parameters in the cost function that are affected by the type of bounding volume are N_b, N_p, and C_b. A tight-fitting bounding volume type, such as the OBB, results in a low N_b and N_p, but has a relatively high C_b, whereas an AABB will result in more tests being performed, but the value of C_b will be lower.

In recent work [46], AABB trees have been shown to yield worse performance than OBB trees for rigid models. However, we will present a

way to speed up overlap testing between relatively oriented boxes of a pair of AABB trees. This results in performance for the AABB tree that is close to the OBB tree's performance for collision detection of rigid models.

5.2.4 AABB Trees vs. OBB Trees

The AABB tree that we consider is, as the OBB tree described in [46], a binary tree. The two structures differ with respect to the freedom of placement of the bounding boxes: AABBs are aligned to the axes of the model's local coordinate system, whereas OBBs can be arbitrarily oriented. The added freedom of an OBB is gained at a considerable cost of storage space. An OBB is represented using 15 scalars, whereas an AABB only requires 6 scalars (for position and extent). Hence, an AABB tree of a model requires roughly half as much storage space as an OBB tree of the same model.

Both tree types are constructed top-down, by recursive subdivision. At each recursion step, the smallest bounding box of the set of primitives is computed, and the set is split by ordering the primitives with respect to a well-chosen partitioning plane. This process continues until each subset contains one element. Thus, a bounding box tree for a set of n primitives has n leaves and $n - 1$ internal nodes.

At each recursion step, we choose the partitioning plane orthogonal to the longest axis of the bounding box. In this way, we get a 'fat' subdivision. In general, fat boxes, i.e., cube-like rather than oblong, yield better performance in intersection testing, since under the assumption that the boxes in a tree mutually overlap as little as possible, a given query box can overlap fewer fat boxes than thin boxes.

We position the partitioning plane along the longest axis, by choosing δ, the coordinate on the longest axis where the partitioning plane intersects the axis. We then split the set of primitives into a negative and positive subset corresponding to the respective halfspaces of the plane. A primitive is classified as positive if the midpoint of its projection onto the axis is greater than δ, and negative otherwise. Figure 5.8 shows a primitive that straddles the partitioning plane depicted by a dashed line. This primitive is classified as positive. It can be seen that by using this subdivision method, the degree of overlap between the AABBs of the two subsets is kept small.

For choosing the partitioning coordinate δ we tried several heuristics. Our experiments with AABB trees for a number of polygonal models show that, in general, the best performance is achieved by simply choosing δ to be the median of the AABB, thus splitting the box in two equal halves.

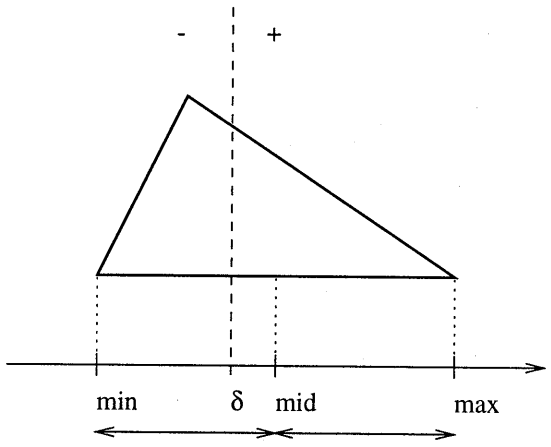


Figure 5.8: The primitive is classified as positive, since its midpoint on the coordinate axis is greater than δ.

Using this heuristic, it may take $O(n^2)$ time in the worst case to build an AABB tree for n primitives, however, in the usual case where the primitives are distributed more or less uniformly over the box, building an AABB tree takes only $O(n \log n)$ time.

Other heuristics we have tried, that didn't perform as well, are: (a) subdividing the set of primitives in two sets of equal size, thus building an optimally balanced tree, and (b) building a halfbalanced tree, i.e., the larger subset is at most twice as large as the smaller one, and the overlap of the subsets' AABBs projected onto the longest axis is minimized.

Occasionally, it may occur that all primitives are classified to the same side of the plane. This will happen most frequently when the set of primitives contains only a few elements. In this case, we simply split the set in two subsets of (almost) equal size, disregarding the geometric location of the primitives.

Building an AABB tree of a given model is faster than building an OBB tree for that model, since the estimation of the best orientation of an OBB for a given set of primitives requires additional computations. We found that building an OBB tree takes about three times as much time as building an AABB tree.

Intersection Testing

Since for both tree types the boxes that are tested for intersection may be arbitrarily oriented, we need an overlap test for relatively oriented boxes.

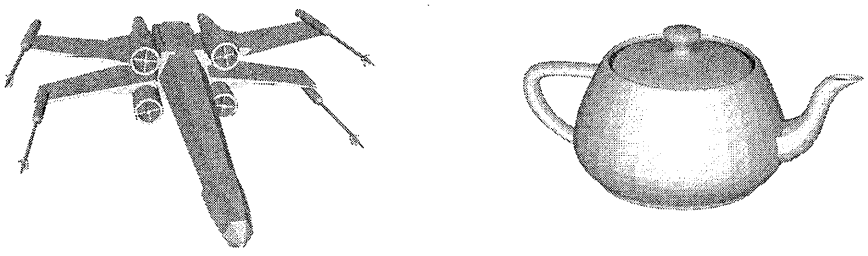
A fast overlap test for oriented boxes is presented by Gottschalk in [46]. We will refer to this test as the **separating-axes test** (SAT). A separating axis of two boxes is an axis for which the projections of the boxes onto the axis do not overlap. The existence of a separating axis for a pair of boxes sufficiently classifies the boxes as disjoint. As we saw in Chapter 4, for any disjoint pair of convex three-dimensional polytopes a separating axis can be found that is either orthogonal to a facet of one of the polytopes, or orthogonal to an edge from each polytope [45]. This results in 15 potential separating axes that need to be tested for a pair of oriented boxes (3 facet orientations per box plus 9 pairwise combinations of edge directions). The SAT exits as soon as a separating axis is found. If none of the 15 axes separate the boxes, then the boxes overlap.

We refer to the original paper [46] for details on how the SAT is implemented such that it uses the least number of operations. For the following discussion, it is important to note that this implementation requires the relative orientation represented by a 3×3 matrix, and its absolute value, i.e., the matrix of absolute values of matrix elements, to be computed before performing the 15 axes tests.

In general, testing two AABB trees for intersection requires more box overlap tests than testing two OBB trees of the same models, since the smallest AABB of a set of primitives is usually larger than the smallest OBB. However, since each tested pair of boxes of two OBB trees normally has a different relative orientation, the matrix operations for computing this orientation and its absolute value are repeated for each tested pair of boxes, whereas for AABB trees the relative orientation is the same for each tested pair of boxes, and thus needs to be computed only once. Therefore, the performance of an AABB tree might not be as bad as we would expect.

In order to compare the performances of the AABB tree and the OBB tree, we have conducted an experiment, in which a pair of models were placed randomly in a bounded space and tested for intersection. The random orientations of the models were generated using the method described by Shoemake in [86]. The models were positioned by placing the origin of each model's local coordinate system randomly inside a cube. The probability of an intersection is tuned by changing the size of the cube. For all tests, the probability was set to approximately 60%.

For this experiment we used Gottschalk's RAPID package [44] for the OBB tree tests. For the AABB tree tests, we used a modified RAPID, in which we removed the unnecessary matrix operations. We experimented with three models: a torus composed of 5000 triangles, a slenderly shaped *X-wing* space craft composed of 6084 triangles, and the archetypical teapot model composed of 3752 triangles, as shown in Figure 5.9. Each run per-



(a) X-wing (b) Teapot

Figure 5.9: Two models that where used in our experiments

forms 100,000 random placements and intersection tests, resulting in approximately 60,000 collisions for all tested models. Table 5.4 shows the results of the tests for both the OBB tree and the AABB tree. The tests were performed on a Sun UltraSPARC-I (167MHz), compiled using the GNU C++ compiler with '-O2' optimization.

The results show that, an AABB tree requires approximately twice as much box intersection tests as an OBB tree, however, the time used for intersection testing is in two cases only 50% longer for AABB trees. The exception here is the torus model, for which the AABB tree uses almost three times as much time as the OBB tree. Apparently, the OBB tree excels in fitting models that have a smooth surface composed of uniformly distributed primitives. Furthermore, we observe that, due to its tighter fit, the OBB tree requires much fewer triangle intersection tests (less than two triangle intersection tests per placement, for the torus and the teapot).

For both tree types, the most time consuming operation in the intersection test is the SAT, so let us see if there is room for improvement. We found that, in the case where the boxes are disjoint, the probability of the SAT exiting on an axis corresponding to a pair of edge directions is about 15%. Figure 5.10 shows a distribution of the separating axes on which the SAT exits for tests with a high probability of the models intersecting. Moreover, for both the OBB and the AABB tree we found that about 60% of all box overlap tests resulted in a positive result. Thus, if we remove

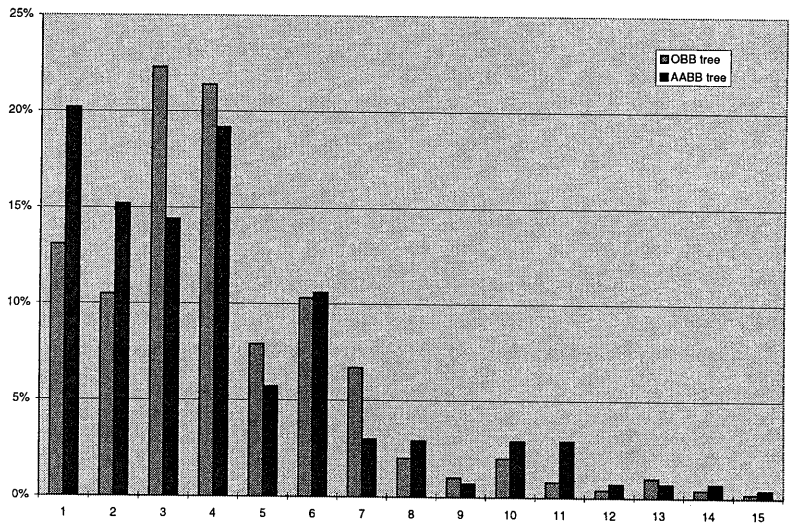


Figure 5.10: Distribution of axes on which the SAT exits in case of the boxes being disjoint. Axes 1 to 6 correspond to the facet orientations of the boxes, and axes 7 to 15 correspond to the combinations of edge directions.

OBB tree							
Model	N_b	C_b	T_b	N_p	C_p	T_p	T_{total}
Torus	10178961	4.9	49.7	197314	15	2.9	52.6
X-wing	48890612	4.6	223.8	975217	10	10.2	234.0
Teapot	12025710	4.8	57.6	186329	14	2.7	60.3
AABB tree							
Model	N_b	C_b	T_b	N_p	C_p	T_p	T_{total}
Torus	32913297	3.7	122.3	3996806	7.2	28.7	151.0
X-wing	92376250	3.1	288.8	8601433	7.1	61.3	350.1
Teapot	25810569	3.3	84.8	1874830	7.4	13.9	98.7

Table 5.4: Performance of the AABB tree vs. the OBB tree, both using the SAT. N_b and N_p are respectively the total number box and triangle intersection tests, C_b and C_p the per-test times in microseconds for respectively the box and triangle intersection test, $T_b = N_b * C_b$ is the total time in seconds spent testing for box intersections, $T_p = N_p * C_p$ is the total time used for triangle intersection tests, and finally T_{total} is the total time in seconds for performing 100,000 intersection tests.

from the SAT the nine axis tests corresponding to the edge directions, we will get an incorrect result only 6% (40% of 15%) of the time.

Since the box overlap test is used for quick rejection of subsets of primitives, exact determination of a box overlap is not necessary. Using a box overlap test that returns more overlaps than there actually are, results in more nodes being visited, and thus more box overlap and primitive intersection tests. Testing fewer axes in the SAT reduces the cost of a box overlap test, but increases the number of box and primitive pairs being tested. Apparently, there is a trade-off of per-test cost against number of tests, when we use a SAT that tests fewer axes.

In order to examine whether this trade-off is in favor of the performance, we repeated the experiment using a SAT that tests only the six facet orientations. We refer to this test as the *SAT lite*. The results of this experiment are shown in Table 5.5. We see a performance increase of about 15% on average for the AABB tree, whereas the change in performance for the OBB tree is only marginal.

We found that the AABB tree's performance benefits from a cheaper but sloppier box overlap test in all cases, whereas the OBB tree shows hardly any change in performance. This is explained by the higher cost of a box overlap test for the OBB tree due to extra matrix operations.

OBB tree							
Model	N_b	C_b	T_b	N_p	C_p	T_p	T_{total}
Torus	13116295	3.7	47.9	371345	12	4.4	52.3
X-wing	65041340	3.4	221.4	2451543	9.3	22.9	244.3
Teapot	14404588	3.5	50.8	279987	13	3.5	54.3
AABB tree							
Model	N_b	C_b	T_b	N_p	C_p	T_p	T_{total}
Torus	40238149	2.4	96.1	5222836	7.4	38.4	134.5
X-wing	121462120	1.9	236.7	13066095	7.0	91.3	328.0
Teapot	30127623	2.1	62.5	2214671	7.0	15.6	78.1

Table 5.5: Performance of AABB tree vs. OBB tree, both using the SAT lite

We see that despite our efforts to improve the performance of the intersection test for AABB trees, OBB trees still beat AABB trees when it comes to intersection testing times for rigid models. Although, construction times and storage usage are in favor of the AABB tree, we regard these issues to be of less importance in our application domain. However, we will present a new scheme for intersection testing of deformable models, for which the AABB tree is found to be the data structure of choice.

5.2.5 AABB Trees and Deformable Models

AABB trees lend themselves quite easily to be used for deformable models. In this context, a deformable model is a set of primitives in which the placements and shapes of the primitives within the model's local coordinate system change over time. A typical example of a deformable model is a triangle mesh in which the local coordinates of the vertices are time-dependent.

Instead of rebuilding the tree after a deformation, it is usually a lot faster to refit the boxes in the tree. The following property of AABBs allows an AABB tree to be refitted efficiently in a bottom-up manner. Let S be a set of primitives and S^+, S^-, subsets of S such that $S^+ \cup S^- = S$, and let B^+ and B^- be the smallest AABBs of respectively S^+ and S^-, and B, the smallest AABB enclosing $B^+ \cup B^-$. Then, B is also the smallest AABB of S. This property is illustrated in Figure 5.11. Of all bounding volume types we have seen so far, AABBs share this property only with DOPs, and does not hold for OBBs.

This property of AABBs yields a straightforward method for refitting

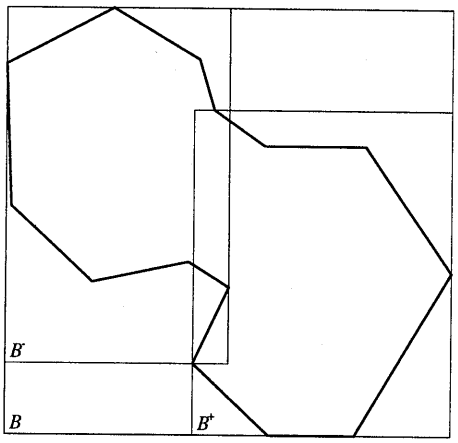
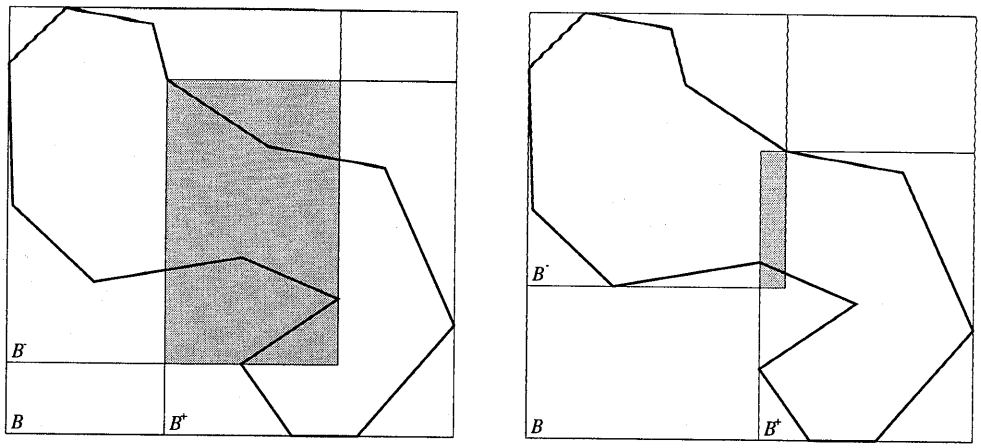


Figure 5.11: The smallest AABB of a set of primitives encloses the smallest AABBs of the subsets in a partition of the set.

a hierarchy of AABBs after a deformation. First the bounding boxes of the leaves are recomputed, after which each parent box is recomputed using the boxes of its children in a strict bottom-up order. This operation may be implemented as a postorder tree traversal, i.e., for each internal node, the children are visited first, after which the bounding box is recomputed. However, in order to avoid the overhead of recursive function calls, we propose a different implementation.

In our implementation the leaves and the internal nodes of an AABB tree are allocated as arrays of nodes. We are able to do this, since the number of primitives in the model is static and a priori known. Furthermore, the tree is built such that each internal child node's index number in the array is greater than its parent's index number. In this way, the internal nodes are refitted properly by iterating over the array of internal nodes in reversed order. Since refitting an AABB takes constant time for both internal nodes and leaves, an AABB tree is refitted in time linear in the number of nodes. Refitting an AABB tree of a triangle mesh takes less than 48 arithmetic operations per triangle. Experiments have shown that for models composed of over 6000 triangles, refitting an AABB tree is about ten times as fast as rebuilding it.

There is, however, a drawback to this method of refitting. Due to relative position changes of primitives in the model after a deformation, the boxes in a refitted tree may have a higher degree of overlap than the boxes in a rebuilt tree. Figure 5.12 illustrates this effect for the model in Figure 5.11. A higher degree of overlap of boxes in the tree results in more



(a) Refitted (b) Rebuilt

Figure 5.12: Refitting vs. rebuilding a model after a deformation

nodes being visited during an intersection test, and thus, worse performance for intersection testing.

We observe a higher degree of overlap among the boxes in a refitted tree mostly for radical deformations such as excessive twists, features blown out of proportion, or extreme forms of self-intersection. However, for deformations that keep the adjacency relation of triangles in a mesh intact, i.e., the mesh is not torn up, we found no significant performance deterioration for intersection testing, even for the more severe deformations. This is due to the fact that the degree of overlap increases mostly for the boxes that are maintained high in the tree, whereas most of the boxes that are tested are the ones that are maintained close to the leaves.

We ran some tests to see how the time used for refitting an AABB tree for a deformable model compares to the intersection testing time. We found that on our testing platform, refitting a triangle mesh composed of a large number (> 1000) of triangles takes 2.9 microseconds per triangle. For instance, for a pair of models composed of 5000 triangles each, refitting takes 29 milliseconds, which is more than 10 times the amount of time it takes to test the models for intersection. Hence, refitting is likely to become the bottleneck if many of the models in a simulated environment are deformed and refitted in each frame. However, for environments with many moving models, in which only a few are deformed in each frame, refitting will not take much more time in total than intersection testing.

Operation	Torus	X-wing	Teapot
(Re)build an OBB tree	0.35s	0.46s	0.27s
Build an AABB tree	0.11s	0.18s	0.08s
Refit an AABB tree	15ms	18ms	11ms
Test a pair of OBB trees	0.5ms	2.3ms	0.6ms
Test a pair of AABB trees	1.3ms	3.3ms	0.8ms

Table 5.6: Comparing the times for a number of operations

In comparison to a previous algorithm for deformable models presented in [88], the algorithm presented here is expected to perform better for deformable models that are placed in close proximity. For these cases, both algorithms show a time complexity that is roughly linear in the number of primitives. However, our approach has a smaller constant (asymptotically 48 arithmetic operations per triangle for triangle meshes). Moreover, our algorithm is better suited for collision detection among a mix of rigid and deformable models, since it is linear in the number of primitives in the deformable models only.

We conclude with a comparison of the performance of the AABB tree vs. the OBB tree for deformable models. Table 5.6 presents an overview of the times we found for operations on the two tree types. We see that for deformable models, the OBB's faster intersection test is not easily going to make up for the high cost of rebuilding the OBB trees, even if only a few of the models are deformed. For these cases, AABB trees, which are refitted in less than 5% of the time it takes to rebuild an OBB tree, will yield better performance, and are therefore the preferred method for collision detection of deformable models.

Chapter 6

Design of SOLID

"Our way is not soft grass. It's a mountain path with lots of rocks."
Ruth Westheimer

In this chapter we discuss the design of SOLID, which is an acronym for Software Library for Interference Detection. We discuss the goals and constraints involved in the design of SOLID, and give a brief description of its functionality. The major design decisions are motivated, and some implementation details are discussed. We evaluate the current version of SOLID (version 2.0) with respect to the goals that were set out. Finally, we discuss the differences between version 1.0 and 2.0 of SOLID, and look into some C++ coding details.

6.1 Requirements

Our goal is to design a general-purpose collision detection library for interactive 3D computer graphics. The library should provide a simple, yet versatile interface to the application program. Furthermore, in order to facilitate remote collision detection, i.e., performing collision detection on a different computer than the one that executes the application program, the **application program interface** (API) should allow low-bandwidth data exchange between the application program and the library. We often refer to an application program as **client**, expressing this remote processing concept.

Remote collision detection may be applied to release the computer on which the application program runs from the task of performing collision detection. This may be done in order to gain performance in other tasks. Tasks that require a lot of computations, such as rendering, compete with

collision detection for processor and/or bus-access time, and may therefore benefit from remote collision detection.

We aim at providing collision detection for all shapes and motions described using VRML [8]. The major issues that need to be addressed in order to achieve compliance with VRML are:

- Models are built using VRML primitive shapes, such as boxes, cones, cylinders, spheres, and complex shapes, represented by soups of points, line segments, and polygons.

- Objects are instances of shapes, i.e., a shape may be used to instantiate multiple objects. This requirement captures the DEF/USE mechanism of VRML.

- The types of movement should be as general as possible. Mostly used are rigid motions (translations and rotations), but also deformations of complex shapes are possible in VRML. In order to allow instances of a shape at different scales, we also need to support nonuniform scalings.

Furthermore, the library should allow the collision handling to be defined by the application program and should compute different types of response data depending on the needs of the application program. In particular, response data which is used for physics-based simulations, such as approximations of contact points and a contact plane, should be computed by the library.

We also address issues such as performance, accuracy, storage usage, and versatility, which express quality, rather than functionality. The following design constraints determine the usefulness of the library for general purposes, and thus, should receive much attention in the design.

- The library should perform collision detection of complex environments at interactive rates. We aim at performance that would allow ten to twenty moving objects composed of thousands of primitives to be tested for collisions in a few hundredths of a second on current high-performance workstations.

- The library should detect collisions accurately. This requires that the configurations of objects processed by the library are identical to the configurations maintained by the client, and that the used algorithms for intersection testing are sufficiently accurate.

- The library should not use too much storage. What is considered too much is a rather fuzzy topic. As a rule of thumb we impose that the amount of storage used by the library is asymptotically linear in the number of primitives in the shape representations, with a constant factor that is roughly a few hundred bytes per primitive.

- The library should not impose constraints on the data structures that are used by the client. The client should be free in the choice of shape representations used for the different tasks. In particular, the library should allow the client to choose a shape representation for collision detection that differs from the representation used for other tasks, such as rendering. This is useful for scaling down the accuracy of the collision detection in favor of the performance by using simpler shapes. Also, if we know a priori that parts of a shape will never collide with any object, for instance the hands of a watch will never collide with objects outside the watch, we may leave out these parts in the shape representation for collision detection.

6.2 Overview of SOLID

This section provides an overview of the SOLID framework. The design decisions are motivated in Section 6.3.

SOLID is a true **software library**, i.e., a collection of functions that can be linked to and used by application programs. The functions are referred to as **commands**. The names and types of these commands, as well as the data types that are used as input and output for these commands, comprise the API. SOLID provides an API conform the C programming language, although the library itself is coded in C++. See Appendix B for a complete description of the commands and data types in the SOLID API. The commands of SOLID fall into four categories:

Shape definition and deformation Commands for defining and deforming shapes relative to a local coordinate system.

Object creation and motion An object is an instance of a shape. Objects are placed or moved by setting or changing the placement of its local coordinate system.

Response definition Collision handling is defined by the client by means of callback functions.

Global controls This category includes commands for performing collision tests, for toggling options that control performance and storage usage, and for tuning tolerances that determine the precision of certain algorithms.

SOLID maintains a separate shape representation for collision detection. Shapes are referred to by pointers to the shape structures maintained by SOLID. These pointers are used exclusively as shape references, thus the client can not access these structures. The shapes are built using commands similar to OpenGL. All commands and data types defined in the SOLID API are prefixed with `dt` for commands, `Dt` for types, and `DT_` for constants. For instance, a pyramid may be built as follows.

```
DtShapeRef pyramid = dtNewComplexShape();

dtBegin(DT_SIMPLEX);
dtVertex(1.0, 0.0, 1.0);
dtVertex(1.0, 0.0, -1.0);
dtVertex(-1.0, 0.0, -1.0);
dtVertex(0.0, 1.27, 0.0);
dtEnd();

dtBegin(DT_SIMPLEX);
dtVertex(1.0, 0.0, 1.0);
dtVertex(-1.0, 0.0, 1.0);
dtVertex(-1.0, 0.0, -1.0);
dtVertex(0.0, 1.27, 0.0);
dtEnd();

dtEndComplexShape();
```

Here, the pyramid shape is composed of a pair of tetrahedra.

The shape types currently supported by SOLID include the primitive shapes: Box, Sphere, Cone, and Cylinder, as used in VRML, as well as complex shapes composed of polytopes. Here, a polytope is a simplex (point, line segment, triangle, tetrahedron), a convex polygon, or a convex polyhedron. No constraints concerning the topology of polytopes in a complex shape are imposed. For instance, a set of polygons need not define a closed surface. We refer to complex shapes defined in this way as **polytope soups**, in accordance with the terminology used by Gottschalk [46].

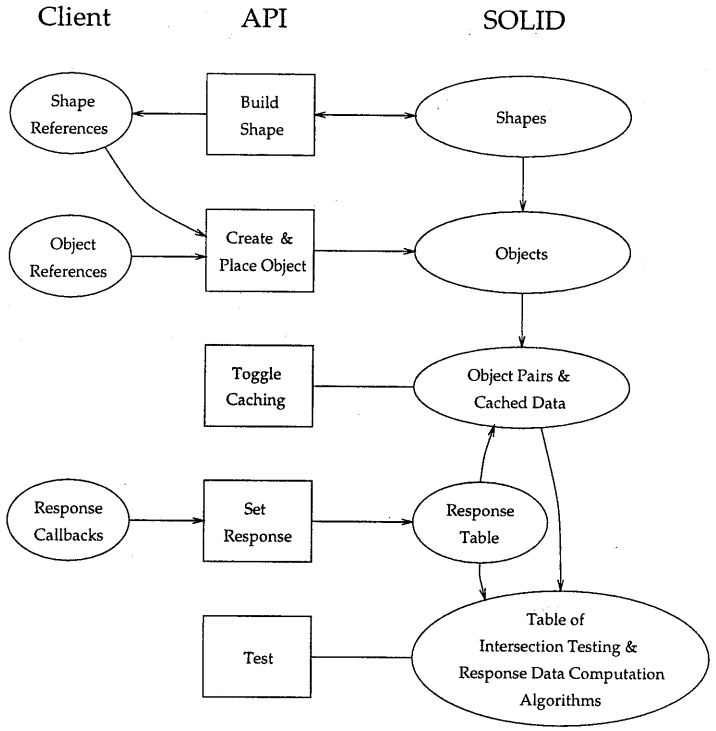


Figure 6.1: A diagram of the SOLID framework. Ellipses denote data structures, boxes denote API commands, and arrows denote data dependencies.

An object is an instance of a shape defined relative to a local coordinate system. A shape may be used to instantiate multiple objects. Note that each object maintains a reference to a shape structure, rather than a copy of the shape representation. In this way, we mimic VRML's DEF/USE mechanism. We refer to objects, using pointers to structures maintained by the client, which are associated with the object. These object references provide easy access to object related data maintained by the client for collision handling. Figure 6.1 shows a diagram of the SOLID framework.

Again, we use commands similar to OpenGL for the placement of objects. However, contrary to OpenGL, we do not have to specify placements for all objects in each frame, since SOLID maintains a representation of each object's placement. Only the placements of objects that are moved with respect to the previous frame are specified.

A change of placement of an object is specified in the following way. SOLID performs transformations only on the object referenced by a state

variable representing the **current object**. The current object is by default the object that is created last. For setting the current object to another object we use the following command

```
dtSelectObject(void * object);
```

Here, `object` is an object reference. Any transformation that follows this command is applied to the object referred to by `object`.

Objects are placed using translations, rotations, and nonuniform scalings of their local coordinate systems. Each transformation is defined relative to the current placement of the local coordinate system, rather than relative to the world coordinate system. However, we may specify a placement relative to the world coordinate system, by first setting the local coordinate system equal to the world coordinate system (loading the identity transformation), and then applying the transformations. The following sample code demonstrates how objects are created and placed relative to the world coordinate system.

```
MyObject khufu;
MyObject khafre;

dtCreateObject(&khufu, pyramid);
dtCreateObject(&khafre, pyramid);

dtSelectObject(&khufu);
dtLoadIdentity();
dtTranslate(248.0, 0.0, 312.0);
dtScale(115.0, 115.0, 115.0);
```

Furthermore, contrary to OpenGL, rotations are specified using quaternions, rather than axis-angle descriptions. We will see more on quaternions in Section 6.3.

Collisions are handled by means of callback functions. A **callback function** is a function that is part of the client, and is called by the library. Response callbacks are functions of the type

```
void (*)(void *client_data,
         void *object1,
         void *object2,
         const DtCollData *coll_data)
```

Here, `client_data` is a pointer to an arbitrary structure maintained by the client, `object1` and `object2` are object references, pointing also to

structures maintained by the client, and `coll_data` is the response data computed by SOLID. For example, a client that counts the number of times two given objects collide would use a callback similar to

```
void collide(void *client_data,
             void *object1,
             void *object2,
             const DtCollData *coll_data) {
  (*(int *)client_data)++;
  printf("Object %d and %d collide\n",
         (*(MyObject *)object1).id,
         (*(MyObject *)object2).id);
}
```

Here, `client_data` points to the counter that maintains the number of collisions.

Responses are defined per pair of objects, for all pairs containing a given object, or as default for all object pairs. A response defined for a pair of objects overrules a response defined for one of the objects in the pair. A response defined for pairs containing a given object overrules the default response. If for each of the objects in a pair a possibly different response is defined, then one of the responses is arbitrarily chosen as the response for the pair. In order to resolve this ambiguity, the client may define the desired response for this pair. The mapping of object pairs to responses is maintained in a structure called **response table**.

For example, suppose we want to write a game called *Naval Warfare*. The game involves ships, of which some are minesweepers, mines, and icebergs. The rules of this game are as follows:

1. If two ships collide then both ships sustain damage.
2. If a ship hits a mine then the ship and the mine are both destroyed.
3. If a ship hits an iceberg then the ship sustains damage.
4. If a mine collides with an iceberg then the mine is destroyed.
5. If a minesweeper hits (detects) a mine, then the mine is destroyed.

We may implement these events using the following responses:

1. Both objects sustain damage.
2. Both objects are destroyed.

3. The object being the ship sustains damage.

4. The object being the mine is destroyed.

As default response we choose Response 1, which implements Rule 1. Rule 2 and 3 are implemented as object responses for respectively mines and icebergs using Response 2 and 3. Response 4 is defined for all pairs containing a mine and an iceberg, and thus overrules the previous two responses. Also, Response 4 is defined for all pairs containing a minesweeper and a mine. We see that with SOLID's response handling mechanism different responses can be applied to different object pairs in a simple manner requiring little overhead.

Optionally, SOLID may perform a pre-sort selecting only the object pairs whose bounding boxes overlap for further processing. For this purpose, SOLID uses bounding boxes that are aligned to the world coordinate system. Each time an object is moved, SOLID dynamically determines a bounding box enclosing the object. For the pre-sort, SOLID applies our adapted version of Baraff's incremental sweep-and-prune algorithm, discussed in Chapter 5. Only the object pairs for which a response is defined are selected. Also, when this option is enabled, SOLID will cache and reuse data from previous intersection tests, such as separating axes, for each selected pair of objects. Caching data may improve performance by exploiting frame coherence.

Finally, the API provides a test command, which performs the collision tests. This command results in response callbacks being called for all colliding pairs of objects, for which a response is defined. The command returns the number of callbacks that are called. Depending on the type of response data required, response data pertaining to the colliding object pairs are computed, and passed to the callbacks as argument.

The algorithms for intersection testing and response data computation are maintained in a table, such that for each pair of shape types, one intersection test and some response data computation algorithms can be defined. In the current SOLID, the choice of algorithms is fixed and can not be altered at run-time by the client.

6.3 Design Decisions

In this section, we motivate the main design decisions concerning SOLID. Here, we will have an inside look in the used algorithms and data structures.

6.3.1 Shape Representation

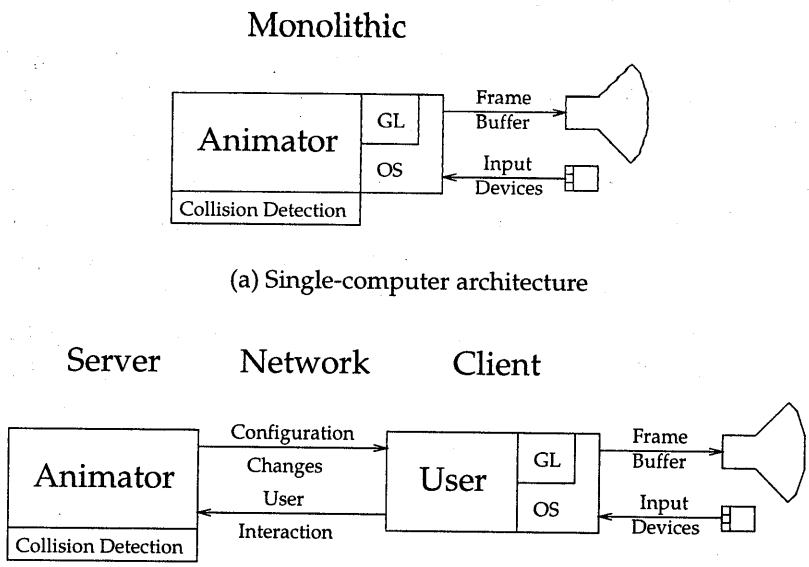
SOLID maintains separate shape representations for performing collision detection, rather than using the shape representations maintained by the client. This construction is chosen for a number of reasons:

- Shape representations used for interactive graphics generally do not allow fast collision detection. Most graphics engines currently used for interactive graphics render shapes represented by polygon soups. These polygonal data are represented in a form that allows low-bandwidth interfacing with, and fast processing by the graphics library, for instance as strips or fans of triangles [102]. Basically, all geometric data is processed anew for each rendered frame. For collision detection, only a small portion of the geometric data needs to be processed for intersection testing in each frame. Spatial data structures, as described in Chapter 5, are used to discard most of the geometric data from intersection testing. Representing shapes as triangle strips is less profitable, since only a few of the triangles in a strip are likely to be inspected during intersection tests. Hence, for collision detection, we opt for a shape representation consisting of single components, maintained in an additional hierarchical data structure, such as a bounding volume hierarchy.

- A design in which the collision detection library uses a shape representation that is maintained by the client inevitably imposes constraints on the client's choice of shape representation. For instance, shapes for which the polygon vertices are computed and processed on the fly, such as quadrics and sweep volumes, can not be used by the collision detection library, since the set of polygons is not explicitly represented by the client.

- In order to keep the bandwidth of data exchange between the client and the library low, which is especially desired in the application of remote collision detection, the collision detection library should maintain its own shape representation.

In multi-user environments simulated across a network, consistent behavior of the environment for all users is achieved by performing the simulations remotely on a single server. This server computes the new configuration of the environment globally for all users. For this purpose, the server performs collision handling. The configuration changes computed by the server are sent to the individual users, where they are processed in order to update the local environment configuration maintained by the



(a) Single-computer architecture

(b) Networked architecture

Figure 6.2: Two environment simulation architectures.

user. Each user renders the updated environment locally according to its own viewpoint, and returns the user interactions to the server. Figure 6.2 shows a networked and a monolithic environment simulation architecture. Obviously, in multi-user applications, using separate shape representations for collision detection and rendering is a necessity rather than an option.

In single-user applications, however, maintaining two sets of geometric data seems rather wasteful. Notably, for polygonal shapes, vertex data takes up the lion's share of the storage used by shape representations, and should desirably not be duplicated. Moreover, shape deformations, which are specified by vertex displacements, result in a performance penalty, in the form of additional vertex copying. Hence, in these cases, a single vertex representation might be more appropriate.

In order to cope with the conflicting demands regarding shape representations for rendering and collision detection, and still keep the vertex data maintained at a single storage location, we provide the following storage method. We borrow a data structure used in OpenGL Version 1.1, called vertex array. A **vertex array**, as the name suggests, is a contigu-

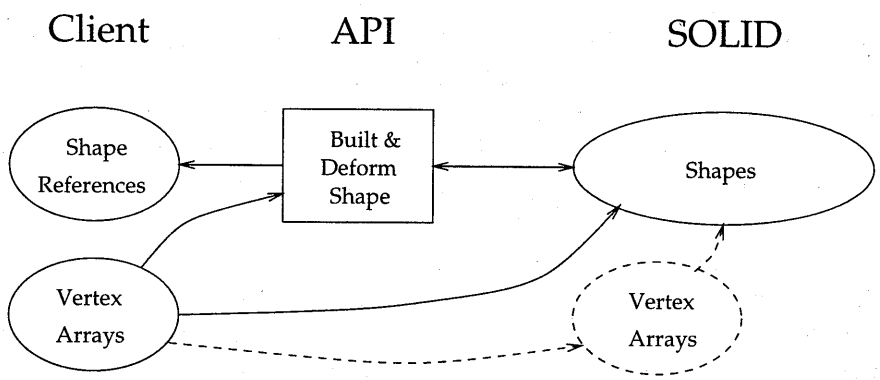


Figure 6.3: Vertex arrays are maintained by the client. SOLID may directly access data maintained in vertex arrays. API commands are used to pass or change the memory location of a vertex array to the shape representation used by the library. In remote collision detection (dashed line), copies of vertex arrays are maintained by the collision detection server.

ous block of vertex data in which each vertex can be randomly accessed using an index. Instead of storing the actual vertices in each of the separate shape representations, we use array indices to refer to the vertices in the array. Since an integer uses less storage than a 3D vertex (4 vs. 24 bytes, using 32 bit integers for indices, and 64 bit floating-point numbers for scalars), sharing vertex data saves a considerable amount of storage. Figure 6.3 illustrates the use of vertex arrays.

Moreover, vertex arrays allow easy interfacing of shape deformations, without the need for copying vertex data. Shape deformations may be specified in two ways: (a) the vertex data in the array of the shape are explicitly changed, after which SOLID is notified of the change, or (b) the current vertex array of the shape is replaced by a vertex array at a different address, i.e., in a different block of memory. Using the latter method, a sequence of key frames of a shape at different stages in the deformation can be stored in memory, and alternated by the client.

The use of vertex arrays conflicts with the requirement that the library should not impose constraints on the shape representation used by the client, and the requirement that the data exchange between the client and the collision detection library should be kept minimal in order to allow efficient collision detection by a remote server. Regarding the first requirement, we note that, although current SOLID imposes rather strict constraints on the arrangement of vertex data in arrays, flexible derefer-

encing methods, for instance as used in OpenGL 1.1 [102], give the client quite a lot of freedom in arranging the vertex data in arrays. Regarding the second requirement, we conclude that for remote collision detection, copying of vertex data is inevitable. However, maintaining vertex data in arrays is still useful, since arrays allow fast exchange of vertex data in blocks of memory.

For fast intersection testing of complex shapes, such as polygon soups, SOLID computes and maintains a bounding volume hierarchy, as discussed in Chapter 5. Although OBB trees have been shown to be faster than AABB trees, we choose to use the latter tree type for the following reasons:

- An AABB tree requires less storage than an OBB tree for the same model. A node in an AABB tree uses 56 bytes (6 scalars + 2 pointers), whereas an OBB tree node uses 128 bytes (15 scalars + 2 pointers). Since for both tree types a tree representation of n primitives has $2n - 1$ nodes, we find that an AABB tree uses asymptotically 112 bytes per primitive, and an OBB tree uses 256 bytes per primitive, excluding the storage taken by the primitives. Considering that for triangle meshes, the per-triangle storage requirements are roughly 40 bytes (vertex data is shared among the triangles in a mesh), we see that the storage used by the tree structures takes up the larger part of the shape representations' storage usage. Hence, for complex models composed of over a 100,000 primitives, the excessive storage usage of the OBB tree might turn out to be a problem. In these cases, the AABB tree, which has a more moderate storage usage, is preferred.

- More critically, OBB trees are not easily adapted to shape deformations. OBB trees need to be rebuilt whenever the shape it represents deforms. AABB trees, on the other hand, can quite easily be adapted to deformations, as discussed in Chapter 5. Refitting an AABB tree is roughly 30 times faster than rebuilding an OBB tree.

Since for intersection testing, AABB trees are not much slower than OBB trees, even for highly concave models at close proximity, we found the AABB tree to be the preferred spatial data structure for complex shapes. Moreover, AABB trees can be built faster than OBB trees, although this feature is not crucial since a bounding volume tree is constructed once in the preprocessing stage. The construction of an AABB tree is performed automatically by SOLID, and does not require instructions by the client. Experiments with different tree building heuristics have shown that the

trees constructed by SOLID are generally close to optimal, hence we consider it unlikely that any hints given by the client on how to perform the model partitioning will result in a significantly faster tree.

6.3.2 Motion Specification

Except for shape deformations, all motions are specified by changing the local coordinate systems of the moving objects. Most commonly used are rigid motions, which are composed of translations and rotations. We translate an object by changing the position of the local origin and rotate an object by changing the orientation of the local basis relative to the world coordinate system. Translations are specified using vectors; rotations are specified using quaternions.

A **quaternion** is a four-dimensional vector. The set of quaternions of length one (points on the 4D unit sphere) map to the set of orientations in three-dimensional space. That is, there is a one-on-one relation between points on a 4D unit hemisphere and orientations in 3D space, since a given quaternion and its negate represent the same orientation.

Quaternions have some benefits over axis-angle orientation representations as used in popular graphics libraries [102]. Most notably, they allow cheap and simple linear interpolations between two orientations [85]. Furthermore, for computing an orthonormal basis whose orientation is given by an axis-angle tuple, we need to evaluate the sine and cosine of the angle, whereas computing a basis given by a quaternion requires only primitive arithmetic operations. Of course, transforming an axis-angle tuple to a quaternion involves sine and cosine evaluations.

However, we prefer quaternions for the following reason. The `sin` and `cos` functions included in the standard C library are computationally quite expensive. Quick and dirty sine/cosine evaluations using lookup tables is faster and may yield reasonable results for most applications. Because enforcing a specific sine/cosine routine deteriorates the versatility, we opt for quaternions instead of axis-angle tuples, since with quaternions calls to sine and cosine routines can be avoided in our library.

Besides rigid motions, we also support nonuniform scalings on local coordinate systems. Nonuniform scalings allow a single shape to be instantiated multiple times, with each instance having different dimensions. Moreover, nonuniform scalings can be used to prevent missing collisions due to sparsely sampled motion for objects that move at high velocities. By scaling a fast moving object with respect to its velocity vector, we can make sure that consecutive configurations of the object overlap, as dis-

cussed in Chapter 2.

We apply a specification method of object placements that differs quite a lot from the one used in OpenGL. OpenGL uses a stack of transformation matrices of which the top matrix represents the current local coordinate system. Matrix stacks are convenient for processing transformation hierarchies. Moreover, OpenGL does not exploit frame coherence, i.e., all objects are processed anew for each frame. We do not consider the use of display lists exploitation of frame coherence, since the use of display lists only saves command calls. It does not reduce the amount of processing. In common usage of matrix stacks, the matrices representing the object placements are created and destroyed in each frame.

For collision detection, however, exploiting frame coherence is crucial in order to attain good performance. Hence, we choose a method for specifying object placements that is persistent in-between frames. Only the object placements that have changed relative to the previous frame, are specified in each frame. The static objects keep their placements from the previous frame.

SOLID does not provide a mechanism for handling transformation hierarchies, such as OpenGL's matrix stacks. However, the transformations computed by OpenGL can be used to specify object placements in SOLID. In OpenGL, affine transformations are represented by 4×4 matrices. The top matrix of the matrix stack in OpenGL can be loaded into an array of 16 scalars and passed to SOLID, which places the current object according to this matrix. In this way, SOLID benefits from the computations performed on OpenGL's matrix stack. Moreover, consistency between the configuration of objects used by OpenGL, and the configuration maintained by SOLID is established quite easily, since both libraries use identical object placements.

6.3.3 Response Handling

Callbacks provide an elegant mechanism for handling response. A callback is a function that handles a specific event, which is the collision of a pair of objects in our case. They are defined by the client, but are not called directly by the client. Instead, a callback is called by the library, whenever a collision is detected for a pair of objects, whose response is handled by the callback. In event-driven systems, such as graphical user interfaces, the use of callbacks for handling all sorts of events is quite common. The idea of using callbacks for response handling in collision detection has been proposed earlier by Zachmann [104].

Some response types require additional data pertaining to the configuration of the intersecting objects. These response data are computed by the library and are passed as argument to the callback functions. Currently, SOLID supports two response data types. The first type is a point common to both objects. A common point is useful when the exact spot where an object hits another object has to be known. In our *Naval Warfare* game, we might want to use this, if the amount of damage a ship sustains depends on the location on its hull, where the ship is hit.

A goal in the design was to provide a response data which can be used for computing reaction forces in order to resolve collisions in physics-based simulations. For this purpose, we need to have an approximation of a points and a contact plane of a pair of colliding objects. In Chapter 2 we saw that these data are best approximated using the frame prior to the collision. The closest points for this frame are taken to be the contact points, and the vector difference of the closest points is a good approximation of the contact plane's normal.

As described in Chapter 4, the preferred method for obtaining the closest points is to back-up to the previous frame on detecting a collision, rather than computing and maintaining the closest points for each simulated frame. Consequently, the library must maintain at all times a representation of the previous frame. A frame is represented by the placements of the local coordinate systems as well as the local coordinates of the vertices of the complex shapes. For the local coordinate systems, we simply cache the last placement for the next frame.

For deformable shapes, we propose a **double-buffering** technique. The local vertex coordinates of a shape for two consecutive frames are stored in two vertex arrays. One array contains the vertex coordinates of the current frame, and the other contains the previous vertex coordinates. In each new frame, the two arrays are swapped, and the new coordinates are stored in the current array. The swapping of arrays is simply done by pointer assignments, hence there is no significant performance loss in applying this technique.

Note that the handling of vertex arrays for double buffering is done by the client. SOLID merely provides a command for changing the pointer to the current vertex array of a complex shape. In each frame, we only cache for each complex shape, the pointer to the current vertex array. It is the responsibility of the client to cache the vertex arrays.

Since it is often necessary to perform multiple collision tests per frame, for instance, for resolving secondary collisions, i.e., collision that result from configuration changes computed by the collision handler, the library needs to be explicitly notified that the simulation proceeds to the next

frame. On receiving this notification, SOLID caches the configuration of the objects in the current frame. This configuration is described by the placements of local coordinate system and vertex arrays. Whenever a closest point pair of a pair of objects needs to be computed, the library restores the configurations of these objects from the previous frame, and performs the closest-point-pair computation.

Note that in order to yield a nonzero plane normal for the contact plane, it is necessary that the objects were not intersecting in the previous frame. Hence, the simulation should never proceed to another frame, if object pairs, for which contact data needs to be computed, are colliding in the current frame. It is the responsibility of the client to guard this condition. For simulations that do not require the use of contact data for any object pair, we do not need to cache previous configurations. Hence, in this case, the client does not need to notify SOLID of a frame change.

6.3.4 Algorithms

When caching is enabled, SOLID maintains a list of object pairs whose bounding boxes overlap. This list is incrementally updated each time an object is moved, using our enhanced version of Baraff's incremental sweep-and-prune algorithm, as described in Chapter 5. The update time per moved object for this algorithm is worst-case $O(n \log k)$, for n objects and a list of k object pairs represented by a balanced binary search tree. However, when frame coherence is high, the update time per moved object is expected to be almost constant.

The bounding boxes used in the incremental sweep-and-prune algorithm are aligned to the world coordinate system. An object's bounding box is recomputed each time the object is moved. The alternative choice of maintaining bounding boxes that have fixed dimensions, is not possible, since objects may also be scaled and deformed. For rigid objects, we could maintain a fixed-size bounding box, large enough to enclose the object in any orientation, however, in SOLID, no distinction is made between rigid and non-rigid objects.

The bounding boxes of objects represented by primitive convex shapes, such as spheres or cones, can be straightforwardly computed. However, for complex shapes composed of thousands of primitives, computing the smallest enclosing box is computationally expensive. Thus, for these shapes we opt for a fast but sloppy solution. Recall that for complex shapes we maintain an AABB tree, which is aligned to the shape's local coordinate system. We take as bounding box of an object that is represented

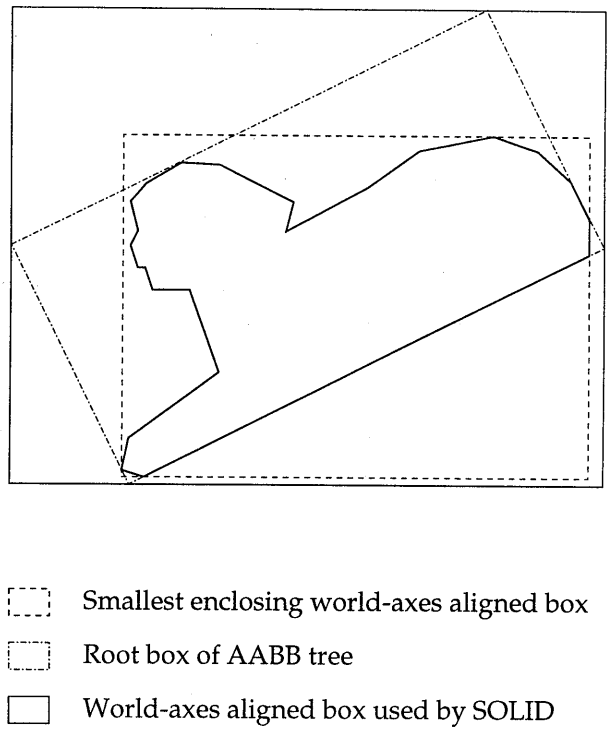


Figure 6.4: Although it is usually larger in size, we use the world-axes aligned bounding box of the AABB tree's root box rather than the smallest world-axes aligned bounding box of a complex shape, since it can be computed much faster.

by a complex shape, the smallest world-axes aligned box that encloses the root box of the AABB tree of the shape. Figure 6.4 illustrates this bounding box computation. As shown in Chapter 5, computing a bounding box of an oriented box takes only 24 arithmetic operations.

The list of object pairs with overlapping bounding boxes is processed each time the test command is issued. When caching is disabled, all pairs of objects for which a response is defined are further processed. Processing an object pair involves testing whether the objects intersect, and if so, computing the specified response data. The intersection testing algorithms, and algorithms for response data computation depend on the shape types of the objects in the pair. For instance, testing a sphere for intersection with a polygon soup, requires a different algorithm than testing a sphere and a cone.

If a the choice of algorithm depends on the dynamic type of only one

of its parameters, we implement the algorithm as a virtual method in C++. However, in our case the algorithms depend on the dynamic type of two data types. This kind of dependency is commonly referred to as **double dispatch** in the object-oriented programming community [65]. Since C++ does not offer a straightforward double-dispatch construction, we chose to apply algorithm tables for implementing double dispatch.

An **algorithm table** is a mapping of pairs of shape types to algorithms defined by C functions. For instance, let the type of the C functions that implement the intersection testing algorithms be given by

```
bool (*)(const Shape&, const Shape&,
         const Transform&, const Transform&);
```

Here, `Shape` is the base type of all shape types. Suppose we have the following C implementation for testing the intersection of a convex and a complex shape.

```
bool intersect(const Convex&, const Complex&,
               const Transform&, const Transform&);
```

Then, the function returned by the algorithm table for the pair of shape types (`Convex`, `Complex`) is the function given by

```
bool intersectConvexComplex(const Shape& shape1,
                            const Shape& shape2,
                            const Transform& xform1,
                            const Transform& xform2)
{
  return intersect((const Convex&)shape1,
                   (const Complex&)shape2,
                   xform1, xform2);
}
```

Note that we may safely cast objects of the base class `Shape` to the derived classes `Convex` and `Complex`, since the types of the objects are checked by the algorithm table.

The **dynamic type** of an object, i.e., the type of the object at the time of creation, can be retrieved at run time using the C++ run-time type identification (RTTI) mechanism [91]. However, since the RTTI mechanism was not supported by most compilers at the time we developed SOLID, we chose to give each shape a tag field which is used to identify the shape's dynamic type.

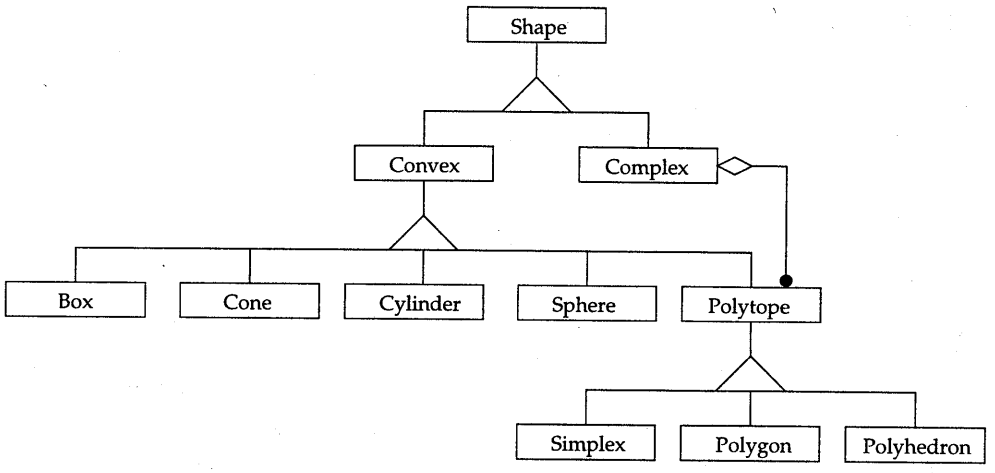


Figure 6.5: An OMT diagram of the class hierarchy of shape types used in SOLID.

In selecting the proper algorithms, SOLID currently discerns two basic shape types, convex shapes and complex shapes. However, the number of basic shape types can easily be increased if necessary for future extensions. New shape types can be added by inserting new entries in the algorithm table. For each pair of shape types that contains the new shape type, we add an intersection testing and, for all response data types, a response data computation algorithm to the algorithm table.

The class of convex shapes is specified into classes for spheres, boxes, cones, cylinders, and polytopes. The class of polytopes again is further specified into simplices, convex polygons, and convex polyhedra. Complex shapes are representations of polytope soups. A complex shape maintains an AABB tree of the set of polytopes. Figure 6.5 shows a diagram of the hierarchy of shape classes used in SOLID.

We test pairs of convex shapes using ISA-GJK, our GJK-based incremental separating-axis computation algorithm discussed in Chapter 4. The common point response data type is also computed using ISA-GJK. The closest points from a previous frame are computed using our improved implementation of the GJK distance algorithm.

Two complex shapes are tested for intersection using the AABB tree intersection test described in Chapter 5. Currently, we use ISA-GJK also for testing pairs of polytopes. For some polytope types, such as triangles, using a dedicated intersection test might be faster. However, polytope intersection tests take only a small portion of the time used for testing a

pair of AABB trees, as we saw in Chapter 5. Hence, the loss of performance is not dramatic when a general approach is taken. The response data for a pair of intersecting complex shapes is the response data computed for the first pair of polytopes that is found intersecting. Currently, SOLID does not process multi-contact collisions.

Finally, for testing a convex shape and a complex shape we apply the following technique. First, we compute a bounding box of the convex shape aligned to the local coordinate system of the complex shape. Overlap tests on aligned bounding boxes are cheap (9 operations), thus, an AABB tree can be quickly traversed depth-first, using the convex shape's bounding box as query volume. On arriving at a leaf node, the polytope maintained in the leaf is tested for intersection with the convex shape and the result is returned. Again, we use GJK based algorithms for primitive intersection testing and response data computation.

When caching is enabled, SOLID caches a separating axis for each disjoint object pair in the list of object pairs updated by the sweep-and-prune algorithm. This axis is used for initializing the ISA-GJK algorithm. In intersection testing on complex shapes, multiple primitive pairs may be tested. Here, a primitive intersection test that is performed after another primitive test uses the separating axis found by this earlier primitive test as initial axis. If none of the primitive intersection tests results in an intersection, the last found separating axis is cached. In this way, coherence between the placements of consecutively tested primitives in a complex shape is exploited.

6.4 Evaluation

In this section we discuss the goals that are attained for SOLID. We also discuss some restrictions that may be removed in future versions.

One major goal was to provide collision detection for all shapes and motions that may be specified using VRML. We achieved compliance with VRML, by incorporating the following features:

- Models can be built using boxes, cones, cylinders, spheres, and complexes of points, line segments and convex polygons.
- Shapes can be instantiated multiple times. This feature captures VRML's DEF/USE mechanism.
- Object placement and motion is specified using translations, rotations, and nonuniform scalings of the object's local coordinate sys-

tems.

- Complex shapes can be deformed.

Note that we did not achieve full compliance, since VRML allows polygons to be non-convex. However, this poses hardly a problem, since the bulk of polygonal models that are currently used in interactive 3D graphics, are composed of convex polygons (mostly triangles). Moreover, code for decomposing a concave polygon in to convex components (triangulation) can easily be obtained [72].

SOLID supports a little more than just the VRML set of primitives, since with SOLID we can specify a shape using convex polyhedra as primitives. So, instead of using the actual shape, we may use the convex hull of a complex shape as a representation for collision detection, when accuracy is less critical. A convex hull representation is sometimes preferable, since it uses far less storage, and results generally in better performance.

Another goal was to incorporate a response data that could be used for physics-based simulations. SOLID is capable of computing an approximation of the contact data, although with some restrictions:

- The client has the responsibility to resolve all collisions in each frame, i.e., the simulation should not proceed to another frame before all object pairs for which contact data is computed are disjoint. Otherwise, SOLID can not compute the normal to the contact plane in the next frames.

- SOLID does not handle multi-contact collisions. If a pair of colliding objects has multiple points of contact, then contact data is computed for only one of the contact points.

Notably, the latter restriction may turn out to be a problem for physics-based collision handlers. Fixing this problem should not pose too many problems, since it merely involves computing the contact data based on all intersecting pairs of primitives of a pair of complex shapes, rather than only on the first intersecting pair of primitives that is found.

Finally, let us examine how well we did with respect to the four quality aspects: performance, accuracy, storage usage, and versatility.

- Performance was our major concern. On our test platform, a Sun UltraSPARC-I (167MHz), testing a pair of complex shapes composed of a few thousands of primitives takes roughly one millisecond. When caching is enabled, exact intersection tests are performed only for object pairs for which the bounding boxes overlap. The cost

of updating the list of object pairs that have overlapping bounding boxes is negligible if frame coherence is high. Suppose that in a real-life setting, the exact intersection test needs to be performed for, say, 10 object pairs per frame. Then, the total time necessary for detecting all collisions would be approximately one hundredth of a second, which is fast enough for interactive applications.

- With respect to the issue of accuracy, we conclude that SOLID performs exact intersection tests and response data computations, within the precision bounds of the used floating-point number representation.

- The amount of storage used by SOLID is quite large, typically in the order of 150 bytes per primitive. Although, this amount is considerably smaller than the amount used by comparable collision detection libraries.

- With respect to versatility, the library imposes few constraints upon the client. All shape and object data is maintained by the library, except for the shape references, which are used for instantiating objects, and the vertex arrays that are used for specifying deformations.

6.5 SOLID Version 1.0

Until now, we have discussed the second version of SOLID that evolved during our research on collision detection methods. The first version of SOLID is in functionality almost a subset of SOLID version 2.0. In this section, we will discuss the main differences, in particular the features that did not make it into the second version.

The first SOLID supports complex shapes composed of polygons (polygon soups) only. Shapes are built using non-convex polygons, and are rigid, i.e., shapes can not be deformed. SOLID 1.0 does not support the use of vertex arrays for defining shapes. Furthermore, SOLID 1.0 allows only rigid motions of objects, thus scaling of an object's local coordinate system is not possible.

In the first version, we use a polygon-polygon intersection test that involves sorting the intersection points along the line of intersection, as discussed in Chapter 3. We sort the points using Quicksort rather than Jordan sorting, since for short point sequences, Quicksort is likely to be faster than Jordan sorting. Moreover, Quicksort implementations are readily available, for instance in the Standard Template Library [70].

In SOLID 2.0, concave polygons may no longer be used as modeling primitives. The reason for this design decision is the following. For testing the intersection of a convex polygon and another convex primitive shape, we can use the ISA-GJK algorithm, as shown in Chapter 4. However, intersection tests between concave polygons and primitive shapes require the use of dedicated algorithms, which are based on the strategy for polygon-volume intersection testing described in Chapter 3. For the sake of simplicity, we decided not to include for each shape type, a dedicated polygon-shape intersection testing algorithm. Instead, we imposed the restriction that polygons have to be convex.

SOLID 1.0 supports as response data, the actual intersection of a pair of objects. Since only polygon soups are used as shapes in SOLID 1.0, the intersection of a pair of objects is a collection of line segments. These line segments are the segments of intersection of the intersecting pairs of polygons. Intersections of coplanar polygons are for convenience's sake ignored. We compute the intersection of two polygons using the algorithm described in Chapter 3, which is an extension of the algorithm used for testing the intersection of polygons.

Initially, it was our intention to approximate a contact point and a contact plane for a pair of colliding objects by applying multivariate analysis on the set of endpoints of these intersection line segments. However, for reasons discussed in Chapter 2 such an approach might fail badly in some cases. Hence, this response data type was dropped in the second version of SOLID.

Finally, in SOLID 1.0, the client may choose between fixed-size boxes and dynamically computed boxes in the sweep-and-prune algorithm. The fixes size boxes are computed once and are large enough to enclose each object in all orientations, whereas the dynamically computed boxes are recomputed each time an object is rotated. Depending on the density of the objects in the scene, either one of these options may result in better performance. We dropped the fixed-size box option, for the obvious reason that for non-rigid objects, a fixed-size box can not be computed a priori, such that the object is enclosed by the box under all possible transformations and deformations.

Overall, version 2.0 of SOLID is more powerful than version 1.0. However, for applications in which only rigid polygonal objects are used, we might prefer SOLID version 1.0 for the following reasons:

- SOLID 1.0 supports the use of non-convex polygons. When polygonal models contain concave polygons, the use of SOLID 1.0 is more convenient.

- SOLID 1.0 computes the intersection of a pair of colliding objects as response data. For some applications, such as CAD and scientific visualization, this response data type may be quite useful.

- SOLID 1.0 may in some occasions be faster than SOLID 2.0. For instance, if the density of objects is low, we can use fixed-size AABBs, instead of dynamically computed AABBs for the sweep-and-prune algorithm. Also, if frame coherence is low, the polygon-polygon intersection test used in SOLID 1.0 is faster than ISA-GJK, as used by SOLID 2.0.

6.6 Implementation Notes

This section is a collection of implementation details concerning the fundamental data structures used in SOLID. Here, we take the opportunity to evangelize on generic programming and the Standard Template Library (STL) [70].

6.6.1 Generic Data Types

Most of SOLID's container data types are implemented using template container classes from the STL. For instance, the set of objects is maintained in an STL map, using the object reference as search key. The list of object pairs, which is maintained when caching is enabled, is implemented as an STL set. Sets and maps are implemented as balanced binary search trees (red-black trees) in STL, and thus allow $O(\log n)$ access and update time for a collection of n elements.

The standard template container classes have proven to be very useful for implementing SOLID's container data structures. The benefit of using standard template classes over hand-coded classes is the fact that the classes are readily available without requiring effort from the programmer to code and test the classes. Also, since all the STL container classes have similar interfaces, changing from one type of container to another is easy.

The use of STL template classes introduces hardly any performance penalty. Most template classes result in code that is as fast as the equivalent hand-coded classes. Moreover, when used properly, STL classes do not require significantly more storage than hand-coded classes would use. However, there are some drawbacks to the use of template classes, although they are quite insignificant.

Since all code in a template class is inlined, each time a template class is instantiated for a given type, executable code is generated by the compiler for all the used inlined code. Hence, the size of the executable code is likely to be larger when template classes are used than when the classes are hand-coded. For SOLID, we managed to keep the total size of the library's executable code close to a modest 200 kilobytes, hence code size wasn't a problem. Less critically, because of the large quantity of inlined code, template classes take longer and require more memory to compile. For the development of SOLID, these drawbacks did not pose any problems.

Furthermore, since STL uses rather advanced template constructions, C++ compilers from a number of vendors currently fail to compile some of the STL classes. With the adoption of the STL in the ISO C++ standard [91], this problem is expected to be solved in the near future.

Besides container classes, the STL also offers a number of template algorithms and functions, such as sort, copy, and search routines. We found the performance of these algorithms to be as good as hand-coded, and better than the standard C library functions. In particular, sorting using the STL sort is significantly faster than using `qsort` from the standard C library.

6.6.2 Fundamental 3D Classes

The fundamental linear algebra classes for representing 3D vectors, points, quaternions, and 3×3 matrices, that were used in SOLID, were developed from scratch. It would be desirable to use existing linear algebra classes for these data types, since implementation of these classes requires considerable effort, and affects the performance of the library to a large extent. However, at the time we started to develop SOLID, no usable C++ linear algebra classes were freely available.

Similar to the STL, our 3D linear algebra classes have all their code inlined. We chose to use inline methods, since they allow better performance than explicitly called functions. Since these methods usually require only a few statements, the increase in executable code that result from using inline methods is insignificant. A number of operations such as vector addition, scalar multiplication, and matrix operations, are denoted by overloaded operators such as + and *=. Overloading of elementary operations for algebraic types greatly enhances the readability of the code.

In SOLID, an affine transformation is represented by a tuple containing a row-major 3×3 matrix as linear component and a vector as translational component. Affine transformations represent local coordinate sys-

tems; the columns of the matrix represent the basis axes and the vector represents the position of the origin relative to a reference coordinate system. We did not decide for a column-major 4×4 matrix representation as used in OpenGL, since we found a row-major 3×3 matrix representation to be more convenient for certain operations. Moreover, manipulations on 4×4 matrices require more computations, for instance, in vector and matrix multiplications.

The transform class contains a method for mapping points from local to reference coordinates. In order to give a transform object the appearance of a function, we chose to implement this method as a function operator. This function operator may be implemented in the following way:

```
Transform::operator()(const Point& p) const {
  return basis * p + origin;
}
```

and may be used as follows,

```
Point world = xform(local);
```

where `xform` is a transform object, and `local` a point given in local coordinates with respect to `xform`.

Also, using the function operator in this way allows easy application of STL algorithms. For instance, we map an array `local_array` of `n` points in local coordinates to an array `world_array` in world coordinates in the following way, using STL's `transform` algorithm :

```
transform(&local_array[0], &local_array[n],
          &world_array[0], xform);
```

These type of constructions are easy to program, easy to read, and result in fast code.

As a conclusion, we present some recommendations for developing code in C++:

- Exploit the STL as much as possible. STL offers container classes and algorithms that result in performance that is, in general, as good as hand-coding.

- Operator overloading for fundamental types is useful, since it improves the readability of the code, and allows easy interfacing with STL algorithms.

- Short functions are best inlined, since the overhead involved with explicit function calls deteriorates performance. However, don't overdo inlining. If the overhead of a function call is small in comparison with the operations performed by the function, inlining yields little performance gain, and increases the size of the executable code if the function is used at multiple places in the code.

Chapter 7

Conclusion

"A movement is accomplished in six stages, and the seventh brings return."
Syd Barrett

In this final chapter we summarize the results of our research, and present some pointers to interesting topics for future work.

7.1 Contributions

Our research in collision detection methods was mostly motivated by the development of SOLID, and has a strong emphasis on practical issues. However, we also contributed work that may have some theoretical interest.

We found new time bounds for both the problem of detecting and of computing the intersection of a pair of three-dimensional non-convex polygons that lie in non-parallel planes. In Chapter 3 we presented algorithms for these problems that run in time linear in the number vertices of the polygons.

7.1.1 Algorithms for Convex Objects

Our main work involves research on algorithms for detecting intersections between convex objects. We examined in detail the Chung-Wang (CW) algorithm [20] and the Gilbert-Johnson-Keerthi (GJK) algorithm [40]. The CW algorithm incrementally computes a separating axis for a pair of disjoint polytopes, whereas the GJK algorithm is used for computing the distance between a pair of convex objects.

The two algorithms perform the computations in a similar way. They are both iterative methods that approximate the required solutions. In each iteration, both algorithms find a better approximation using a pair of support points of the objects. However, despite their similarities, the algorithms perform quite differently.

The major benefit of the CW algorithm is the fact that it is incremental, i.e., if the algorithm is initialized by an axis that is close to being a separating axis, then the algorithm needs much fewer iterations to find a proper separating axis than if started from scratch. This is useful in computer animation, where there is usually a lot of frame coherence. By using the separating axis found in a previous frame for computation of a separating axis in the current frame, we may speed-up the collision detection considerably.

However, the CW algorithm also suffers from some drawbacks. In the CW algorithm, each iteration takes time linear in the number of preceding iterations. This should not be a problem if we could keep the number of iterations small. However, we showed in Chapter 4 that for CW, convergence can be extremely slow, and thus the algorithm may require a large number of iterations. Furthermore, due to this slow convergence, or rather, lack of convergence, generalization of CW to a larger class of convex objects, including, for instance, convex quadrics, is not straightforward. Although the CW algorithm seemed very promising at first, we were unable to overcome these problems, and thus, shifted our attention to the GJK algorithm.

The GJK algorithm takes only a constant time per iteration (if we discount the time for computing the support points). Moreover, GJK converges faster, and thus, requires fewer iterations than CW. Also, GJK is applicable to general convex objects [39]. However, for the original GJK, the computation time per iteration is quite large, in particular, much larger than the time taken by one of the first few iterations of the CW algorithm. Furthermore, the original GJK is not incremental, and hence, does not exploit frame coherence. Finally, GJK may suffer from termination problems due to rounding errors in floating-point arithmetics for pairs of polyhedra that differ a lot in size.

In this thesis, we addressed the problems concerning GJK, and presented enhancements of the algorithm. We improved the performance of GJK, by caching computed values in-between iterations, thus significantly reducing the number of computations per iteration. We showed that by modifying the algorithm such that it returns a separating axis, rather than the distance, the number of iterations can be greatly reduced. Moreover, we showed that this modified GJK can be used for incremental separating

axis computation. We solved GJK's termination problems by extending the algorithm such that it detects numerical problems. Experiments have shown that, our improved GJK algorithm, referred to as ISA-GJK, has a performance that is close to CW's, and is significantly faster than the Lin-Canny incremental distance algorithm [61].

7.1.2 Spatial Data Structures

Most collision detection methods described in literature are aimed at reducing the cost of intersection tests for complex models composed of a large number of objects. We discern two types of problems: (a) finding all intersecting pairs of objects among freely moving objects, and (b) finding all intersecting pairs of primitives of two complex shapes composed of thousands of primitives.

For the first problem, we presented an improvement of Baraff's incremental sweep-and-prune scheme [4]. This scheme involves maintaining a list of pairs of objects whose world-axes aligned bounding volumes overlap. The improvement concerns the update time in cases where only a few of the objects are moving at a given time. When frame coherence is high, Baraff's scheme allows updating the list of object pairs in roughly linear time with respect to the total number of objects, whereas with our scheme, an update is performed in time linear in the number of *moving* objects only.

For the second problem, we examined two data structures: the oriented bounding box (OBB) tree, and the axis-aligned bounding box (AABB) tree. Both data structures are bounding volume hierarchies, and are constructed by recursively bipartitioning a set of primitives in two geometrically coherent subsets, and computing bounding volumes for the sets. For testing the intersection between two complex shapes undergoing rigid motion, the OBB tree yields, in general, the best performance of the two data structures. However, the differences in performance are not extreme. We presented an intersection test for a pair of AABB trees that takes, despite the looser fit of the AABBs, only about 50% more time on average than the best implementation for OBB trees, currently available. Furthermore, an AABB tree takes roughly half as much storage as an OBB tree, and takes less time to construct.

Yet, the most significant benefit of AABB trees over OBB trees concerns shape deformations. In Chapter 5 we presented a fast method for updating an AABB tree after a shape deformation. Updating an OBB tree after a shape deformation is more complex, and involves reconstructing the tree for the deformed shape. Hence, we found the AABB tree, as presented

in this thesis to be the data structure of choice for intersection testing of complex deformable shapes.

7.1.3 Collision Detection Library

Our research on collision detection methods was aimed at designing a library for collision detection in interactive 3D computer animation. This work resulted in a library, called SOLID, which is an acronym for Software Library for Interference Detection. SOLID incorporates the following innovative features:

- SOLID supports models composed of a mix of shape types, including VRML's primitive shapes and complex shapes composed of convex polygons and convex polyhedra.
- SOLID supports deformations of complex shapes.
- SOLID allows besides translations and rotations, also nonuniform scalings on objects. This feature is useful for instantiating multiple objects of different dimensions using a single shape.
- SOLID optionally computes response data that represent the approximated contact points and contact plane of a pair of colliding objects. This response data type is useful in physics-based simulations.

SOLID is written in standard C++, however, the library has an API of standard C functions, and can thus be used in both C and C++ applications. The complete source code as well as the documentation for SOLID is freely distributed under the terms of the GNU Library General Public License [35]. Thus far, interest in using SOLID has been shown for wide variety of application areas, including 3D games, haptic interfaces, CAD/CAM, flight and underwater simulators, rapid prototyping, and motion planning.

7.2 Future Work

As in any line of scientific research, the work on collision detection is never done. In this section, we discuss a number of interesting research topics that may be addressed in future work.

7.2.1 Shape Types

Currently, polyhedra are represented using boundary representations consisting of a collection of polygons. For some polyhedra, a representation described by the union of as a set of convex polyhedra may result in better performance and require less storage. Methods for decomposing a concave polyhedron into a minimum number of (possibly overlapping) convex components are still a subject of research.

Implicit surfaces, parametric surfaces (NURBs, Bézier patches), and constructive solid geometry (CSG), are popular shape representations in geometric modeling. Since rendering of these shape representations at interactive rates is not yet feasible, the shapes are usually represented as polyhedral shapes which *can* be rendered interactively on current graphics hardware. However, for the purpose of collision detection, implicit surfaces, parametric surfaces, and CSG representations might be more suitable than polygon soups. Further research is necessary in order to decide which type of shape representation is best for collision detection.

7.2.2 Algorithms

Both the CW algorithm and the GJK algorithm require the computation of support points in each iteration. Computing a support point of a polytope takes in the worst case linear time with respect to the number of vertices. However, by maintaining the adjacency graph of the polytope's vertices, finding a support point takes almost constant time, when frame coherence is high. Using the Dobkin-Kirkpatrick hierarchical polytope representation [27], the worst-case bound for computing a support point can be reduced to $O(\log n)$ for n vertices [20]. An interesting extension of the DK representation would be a polytope representation for which it takes $O(\log n)$ time in worst case, and, when frame coherence is high, constant times for computing a support point. Such a polytope representation should support local search of support points over the boundary of the polytope, similar to the adjacency graph, and should make clever use of the shortcuts provided by the DK representation.

For reducing the number of pairwise object intersection tests among a collection of freely moving objects, we proposed the use of an incremental sweep-and-prune algorithm, which maintains the pairs of objects whose bounding boxes overlap. This algorithm allows update of the list of object pairs in time that is close to being linear in the number of moving objects, when frame coherence is high. Similar time bounds might be achieved by applying a three-dimensional generalization of the fieldtree [34].

In general, OBB trees yield the best performance of all bounding volume hierarchies currently used for testing intersections between complex shapes. It shows that the relatively high cost of testing a pair of OBBs for overlap is largely made up for by its tight fit. In this respect, it may be a good idea to take this one step further.

The convex hull of a set of polytopes is in general smaller than the smallest OBB of the set. However, the cost of testing a pair of convex hulls for overlap is considerably higher than for OBBs. If the cost of testing a pair of convex hulls for overlap is sufficiently small, a hierarchy of convex hulls for a set of polytopes might perform better than an OBB tree.

The cost of overlap testing for convex hulls may be reduced by exploiting coherence. For instance, by exploiting frame coherence, we can reduce the cost of each overlap test to almost constant time, as described in Chapter 4. Moreover, overlap tests on the child hulls of an internal node may repeat some of the computations performed for overlap tests on the parent node's convex hull. For instance, the same support points will often be computed for both the parent and the child hulls. Hence, by caching and reusing these computed values, we may speed-up hull tests for the child nodes, if the hull test for the parent node fails.

Using convex hulls as bounding volumes in volume hierarchies seems promising. However, in order to exploit the different types of coherence, intricate data structures and algorithms are needed. Altogether, we reckon convex hull hierarchies to be well-worth examining further.

Finally, we would like to hint at a topic that has received a lot of attention in the graphics community lately, namely progressive meshes [55]. A **progressive mesh** representation is a way of storing triangle meshes, that allows selective refinement of parts of the mesh. This is useful in interactive graphics, since it allows maintaining at a given time only those parts of a shape in memory, that are actually rendered. This results in higher rendering performance, since fewer triangles need to be processed, a lower storage usage, and a lower transmission bandwidth for networked graphics architectures.

Storage usage and transmission bandwidth are also critical for shape representations used for collision detection. Hence, progressive shape representations of complex models may be an interesting research topic in the context of collision detection as well. For this purpose, the progressive shape representations should include a description of the bounding volume hierarchy used for speeding up intersection testing. The bounding volumes that are traversed during an intersection test can be loaded on demand, i.e., whenever a bounding volume test fails, the bounding volumes of the child nodes are loaded and tested.

Appendix A

Linear Analysis

Although the content of this section is explained similarly, and often more thoroughly, in the bulk of geometry literature, for instance in [23], we still find it useful to include it since it serves as an easy reference and an introduction to the notation used in this thesis.

A.1 Notational Conventions

In this section we establish the notational conventions used throughout this text. The reader is assumed to have a basic grasp of linear algebra and set theory; it is not our objective to provide all the formalities of the mathematical concepts used in this thesis.

The set of real numbers is denoted by $\mathbb{R}$. In the context of vector spaces, real numbers are referred to as **scalars** and denoted by lowercase Greek letters, such as α, β, γ. The vector space of d-tuples $(\alpha_1, \ldots, \alpha_d)$ is denoted by $\mathbb{R}^d$. Elements of $\mathbb{R}^d$ are referred to as **vectors** and denoted by lowercase boldface letters, such as $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$. The **zero vector** is denoted by $\mathbf{0}$.

Matrices over $\mathbb{R}$ are denoted by uppercase boldface letters, such as $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$. The matrix $\mathbf{A} = [\alpha_{ij}]$ denotes the matrix with number α_{ij} in the ith row and jth column. The **transpose** of a matrix $\mathbf{A}$ is denoted by $\mathbf{A}^{\mathrm{T}}$. In matrix notation, vectors are regarded as columns, which are $m \times 1$ matrices. For a set of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^m$ we denote the $m \times n$ matrix with columns $\mathbf{v}_j$ as $[\mathbf{v}_1 \ldots \mathbf{v}_n]$.

We consider only square matrices, i.e., matrices with an equal number of columns and rows. The determinant of a square matrix $\mathbf{A}$ is denoted by $\det(\mathbf{A})$. A matrix is called **singular** iff its determinant is zero, and **nonsingular** otherwise. The set of nonsingular $d \times d$ matrices forms a group, with matrix multiplication as operator. The **identity** is the matrix $\mathbf{I} = [\delta_{ij}]$

where δ_{ij}, referred to as the **Kronecker symbol**, is defined as

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

The inverse of a matrix $\mathbf{A}$ is denoted by $\mathbf{A}^{-1}$.

A set is defined either by enumeration, such as $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, or conditionally, such as $\{\mathbf{x} \in \mathbb{R}^n : P(\mathbf{x})\}$, which is the set of $\mathbf{x} \in \mathbb{R}^n$ for which predicate $P(\mathbf{x})$ is true. Sets are denoted by uppercase italics, such as A, B, C. The empty set is denoted by $\emptyset$. The **power-set** of a set X, denoted by $\mathcal{P}(X)$, is the set of all subsets of X. We will adopt the convention that functions $f : X \to Y$ are silently lifted to $\mathcal{P}(X) \to \mathcal{P}(Y)$ according to $f(A) = \{f(a) : a \in A\}$.

A.2 Vector Spaces

A **linear combination** of n vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ is a vector of the form

$$\mathbf{v} = \alpha_1 \mathbf{v}_1 + \cdots + \alpha_n \mathbf{v}_n.$$

The **span** of a set of vectors is the set of linear combinations of vectors in the set. The span of a single nonzero vector is called an **axis**. A set of vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ is said to be **linearly independent** if the equation

$$\alpha_1 \mathbf{v}_1 + \cdots + \alpha_n \mathbf{v}_n = \mathbf{0}$$

yields $\alpha_1 = \cdots = \alpha_n = 0$. A **basis** of a vector space is a linearly independent set of vectors whose span is the whole space. The number of vectors in the basis is referred to as the **dimension** of the space. For a basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ the equation

$$\mathbf{v} = \alpha_1 \mathbf{b}_1 + \cdots + \alpha_n \mathbf{b}_n$$

has exactly one solution for a given vector $\mathbf{v}$. Hence, $\mathbf{v}$ is uniquely identified by the n-tuple $(\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$ with respect to the given basis. The scalars α_i are called the **vector components** of $\mathbf{v}$ relative to $\{\mathbf{b}_i\}$. In particular, the components of the basis vectors $\mathbf{b}_i$ are

$$\mathbf{b}_i = (\delta_{i1}, \ldots, \delta_{in}).$$

A **linear transformation** is a function $\mathbf{T}$ that maps vectors to vectors according to

$$\mathbf{T}(\alpha \mathbf{v} + \beta \mathbf{w}) = \alpha \mathbf{T}(\mathbf{v}) + \beta \mathbf{T}(\mathbf{w}).$$

Consequently, a linear transformation is determined by the image of the basis. We consider only linear transformations from a vector space onto itself. For these transformations, the image of a basis is itself a basis. Let $B' = \{\mathbf{b}'_i\}$, be the image of a basis B such that the components of $\mathbf{b}'_i$ are given relative to B. The image of a vector $\mathbf{x} = (\alpha_1, \ldots, \alpha_n)$ relative to B is

$$\mathbf{x}' = \alpha_1 \mathbf{b}'_1 + \cdots + \alpha_n \mathbf{b}'_n.$$

We introduce a matrix $\mathbf{B}' = [\mathbf{b}'_1 \ldots \mathbf{b}'_n]$, and write this equation as

$$\mathbf{x}' = \mathbf{B}'\mathbf{x}.$$

Here, B' is indeed a basis iff $\mathbf{B}'$ is nonsingular. We will often use the same symbol to denote a matrix and the corresponding linear transformation.

A.3 Affine Spaces

An **affine space** consist of a set of **points**, an associated vector space, and two operations: the addition of a point and a vector, and the subtraction of two points. Points are denoted, as vectors, by lowercase boldface letters. The addition of a point and a vector yields a point according to the rules $\mathbf{p} + \mathbf{0} = \mathbf{p}$ and $(\mathbf{p} + \mathbf{v}) + \mathbf{w} = \mathbf{p} + (\mathbf{v} + \mathbf{w})$. Conversely, the subtraction of two points yields a vector according to the rule $\mathbf{p} + (\mathbf{q} - \mathbf{p}) = \mathbf{q}$. Although addition and scalar multiplication are not defined for points, we define an **affine combination** of points $\mathbf{p}_0, \ldots, \mathbf{p}_n$ as

$$\mathbf{p} = \alpha_0 \mathbf{p}_0 + \alpha_1 \mathbf{p}_1 + \cdots + \alpha_n \mathbf{p}_n \quad \text{for } \alpha_0 + \cdots + \alpha_n = 1.$$

This expression makes sense if we are allowed to formally eliminate α_0 and write

$$\mathbf{p} = \mathbf{p}_0 + \alpha_1(\mathbf{p}_1 - \mathbf{p}_0) + \cdots + \alpha_n(\mathbf{p}_n - \mathbf{p}_0),$$

which is obviously a point. The **affine hull** of a set of points A, denoted by $\mathrm{aff}(A)$, is the set of affine combinations of points in A. An **affine set** is a set of points that is closed under affine combinations. A set of points $\{\mathbf{p}_0, \ldots, \mathbf{p}_n\}$ is called **affinely independent** if the set $\{\mathbf{p}_1 - \mathbf{p}_0, \ldots, \mathbf{p}_n - \mathbf{p}_0\}$ is linearly independent. The **dimension** of an affine set is the number of points in any affinely independent set, whose affine hull is the affine set, minus one.

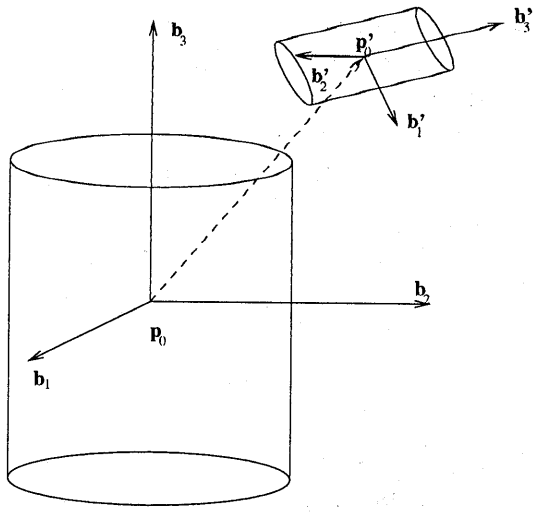


Figure A.1: An affine transformation in $\mathbb{R}^3$

A **coordinate system** is a tuple of a point and a basis. The point is called the **origin** of the coordinate system. For a given coordinate system with origin $\mathbf{p}_0$ and basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, the equation

$$\mathbf{p} = \mathbf{p}_0 + \alpha_1 \mathbf{b}_1 + \cdots + \alpha_n \mathbf{b}_n$$

has exactly one solution for a given point $\mathbf{p}$. The point $\mathbf{p}$ is uniquely identified by the vector $(\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$ with respect to the coordinate system. The components α_i are called the **coordinates** of $\mathbf{p}$. We identify a point with the corresponding vector of coordinates relative to the given coordinate system.

An **affine transformation** is a function $\mathbf{T}$ that maps points to points according to

$$\mathbf{T}(\alpha\mathbf{p} + \beta\mathbf{q}) = \alpha\mathbf{T}(\mathbf{p}) + \beta\mathbf{T}(\mathbf{q}) \quad \text{for } \alpha + \beta = 1.$$

Consequently, an affine transformation is determined by the images of the basis and the origin of the given coordinate system. Let $\mathbf{B}$ represent the image of the basis, and let $\mathbf{c}$ be the image of the origin. The corresponding affine transformation $\mathbf{T}$ is given by

$$\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}.$$

The set of affine transformations from $\mathbb{R}^n$ onto $\mathbb{R}^n$ forms a group with function composition as operator

$$\mathbf{T}_2 \circ \mathbf{T}_1(\mathbf{x}) = \mathbf{B}_2(\mathbf{B}_1\mathbf{x} + \mathbf{c}_1) + \mathbf{c}_2 = \mathbf{B}_2\mathbf{B}_1\mathbf{x} + \mathbf{B}_2\mathbf{c}_1 + \mathbf{c}_2$$

and inverse

$$\mathbf{T}^{-1}(\mathbf{x}) = \mathbf{B}^{-1}(\mathbf{x} - \mathbf{c}) = \mathbf{B}^{-1}\mathbf{x} - \mathbf{B}^{-1}\mathbf{c}.$$

The identity of the group of affine transformations is $\mathbf{I}$.

A **local coordinate system** is a coordinate system that is defined relative to a **parent coordinate system**. This parent coordinate system in turn can be a local coordinate system to yet another coordinate system, and so on. The root of such a hierarchy of relatively defined coordinate system is referred to as the **world coordinate system**.

A coordinate system is defined local to a parent coordinate system by giving the coordinates of its origin and its basis vectors in parent coordinates, i.e., relative to the parent coordinate system. Points given in local coordinates may be transformed to parent coordinates by means of an affine transformation. Let $\mathbf{B} = [\mathbf{b}_1 \cdots \mathbf{b}_n]$, where $\mathbf{b}_i$ are the local basis vectors in parent coordinates, and $\mathbf{c}$ the position of the local origin in parent coordinates. The affine transformation $\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$ maps the local coordinates of a point to parent coordinates. Hence, we can (and do) identify a local coordinate system with the corresponding affine transformation.

A.4 Euclidean Spaces

A Euclidean space is an affine space with a notion of length and distance, defined by means of the **dot product**. The dot product of vectors $\mathbf{v}$ and $\mathbf{w}$, denoted by $\mathbf{v} \cdot \mathbf{w}$, yields a scalar according to the following rules

1. symmetric: $\mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{v}$.
2. bilinear: $\mathbf{u} \cdot (\alpha\mathbf{v} + \beta\mathbf{w}) = \alpha\mathbf{u} \cdot \mathbf{v} + \beta\mathbf{u} \cdot \mathbf{w}$.
3. positive definite: $\mathbf{v} \cdot \mathbf{v} > 0$ for $\mathbf{v} \neq \mathbf{0}$.

Note that these rules do not uniquely determine the dot product.

In order to determine the dot product, we introduce a basis $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$, referred to as the **standard basis**, for which

$$\mathbf{e}_i \cdot \mathbf{e}_j = \delta_{ij}.$$

Within the scope of this thesis we assume that the basis of the world coordinate system is the standard basis.

The **length** of a vector $\mathbf{v}$, denoted by $\|\mathbf{v}\|$, is defined as

$$\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}}.$$

The **distance** between two points $\mathbf{p}$ and $\mathbf{q}$, denoted by $d(\mathbf{p}, \mathbf{q})$ is the length of the vector $\mathbf{p} - \mathbf{q}$;

$$d(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|.$$

The **angle** α between two nonzero vectors $\mathbf{v}$ and $\mathbf{w}$ is defined by

$$\cos(\alpha) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|} \quad 0 \leq \alpha \leq \pi.$$

A pair of vectors $\mathbf{v}$ and $\mathbf{w}$ are said to be **orthogonal**, denoted by $\mathbf{v} \perp \mathbf{w}$, if $\mathbf{v} \cdot \mathbf{w} = 0$. It can be proven that a set of mutually orthogonal nonzero vectors is linearly independent. A basis $\{\mathbf{b}_i\}$ for which $\mathbf{b}_i \cdot \mathbf{b}_j = \delta_{ij}$, as for the standard basis, is called **orthonormal**. For vectors $\mathbf{v}$ and $\mathbf{w}$ relative to an orthonormal basis we find that the dot product is given by

$$\mathbf{v} \cdot \mathbf{w} = \mathbf{v}^{\mathrm{T}} \mathbf{w}.$$

A **Cartesian system** is a coordinate system that has an orthonormal basis.

When we do not care about the length of a vector, we often refer to the vector as **direction**, or **orientation** (for normals to an oriented plane). We may identify a direction or orientation by a **unit vector**, which is a vector of length one. When we merely want to denote the linear subspace spanned by a vector, i.e., we are indifferent about multiplications of the vector by a negative number, we refer to the vector as **axis**. Hence, each axis corresponds to two directions.

A.5 Affine Transformations

The group of affine transformations has a number of important subgroups. We have already seen one of them, namely, the group of linear transformations. The group of **translations** is formed by the transformations

$$\mathbf{T}(\mathbf{x}) = \mathbf{x} + \mathbf{c}.$$

The group of **rotations** about the origin is formed by the transformations

$$\mathbf{R}(\mathbf{x}) = \mathbf{B}\mathbf{x} \quad \text{where } \mathbf{B}^{-1} = \mathbf{B}^{\mathrm{T}} \text{ and } \det(\mathbf{B}) = 1.$$

A matrix $\mathbf{B}$ for which $\mathbf{B}^{-1} = \mathbf{B}^{\mathrm{T}}$ is called **orthogonal**. An orthogonal matrix maps an orthonormal basis to an orthonormal basis (note the nomenclature!), since for orthogonal $\mathbf{B}$ and orthonormal basis $\{\mathbf{b}_i\}$ we have

$$(\mathbf{B}\mathbf{b}_i) \cdot (\mathbf{B}\mathbf{b}_j) = (\mathbf{B}\mathbf{b}_i)^{\mathrm{T}} (\mathbf{B}\mathbf{b}_j) = \mathbf{b}_i^{\mathrm{T}} \mathbf{B}^{\mathrm{T}} \mathbf{B} \mathbf{b}_j = \mathbf{b}_i^{\mathrm{T}} \mathbf{b}_j = \mathbf{b}_i \cdot \mathbf{b}_j = \delta_{ij}.$$

Furthermore, it follows that any matrix that maps an orthonormal basis to an orthonormal basis is necessarily orthogonal.

The group of **rigid motions** in $\mathbb{R}^n$ is the supergroup of translations and rotations. The group of **length-preserving** transformations is formed by the set of affine transformations $\mathbf{T}$ for which

$$\|\mathbf{T}(\mathbf{x}) - \mathbf{T}(\mathbf{y})\| = \|\mathbf{x} - \mathbf{y}\|$$

holds for all points $\mathbf{x}$ and $\mathbf{y}$. An affine transformation $\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$ is length-preserving iff $\mathbf{B}$ is orthogonal, since

$$\|\mathbf{T}(\mathbf{x}) - \mathbf{T}(\mathbf{y})\| = \|\mathbf{B}\mathbf{x} - \mathbf{B}\mathbf{y}\| = \|\mathbf{B}(\mathbf{x} - \mathbf{y})\| = \sqrt{\mathbf{B}(\mathbf{x} - \mathbf{y}) \cdot \mathbf{B}(\mathbf{x} - \mathbf{y})},$$

which is reduced to $\sqrt{(\mathbf{x} - \mathbf{y}) \cdot (\mathbf{x} - \mathbf{y})} = \|\mathbf{x} - \mathbf{y}\|$ iff $\mathbf{B}$ is orthogonal.

A **reflection** in a plane through the origin is an affine transformation of the form

$$\mathbf{W}(\mathbf{x}) = \mathbf{B}\mathbf{x} \quad \text{where } \mathbf{B} \text{ is orthogonal and } \det(\mathbf{B}) = -1.$$

Any length-preserving transformation is either a rigid motion or a composition of a translation and a reflection [23].

The group of **uniform scalings** about the origin is the group of transformations of the form

$$\mathbf{U}(\mathbf{x}) = \alpha\mathbf{x} \quad \text{for } \alpha \neq 0.$$

Compositions of length-preserving transformations and uniform scalings constitute the group of **angle-preserving** transformations. For each angle-preserving transformation $\mathbf{T}$ an $\alpha > 0$ exists, such that for arbitrary points $\mathbf{x}$ and $\mathbf{y}$

$$\|\mathbf{T}(\mathbf{x}) - \mathbf{T}(\mathbf{y})\| = \alpha\|\mathbf{x} - \mathbf{y}\|.$$

The group of **nonuniform scalings** about the origin is the group of transformations of the form

$$\mathbf{S}(\mathbf{x}) = [\alpha_{ij}]\mathbf{x} \quad \text{where } \alpha_{ij} \neq 0 \text{ iff } i = j.$$

Notice that the group of uniform scalings is a subgroup of the group of nonuniform scalings.

As shown in [43], any affine transformation $\mathbf{A}$ can be constructed as a composition of a translation $\mathbf{T}$, two rotations $\mathbf{R}_\mathrm{L}$ and $\mathbf{R}_\mathrm{R}$, and a nonuniform scaling $\mathbf{S}$, such that

$$\mathbf{A} = \mathbf{T} \circ \mathbf{R}_\mathrm{L} \circ \mathbf{S} \circ \mathbf{R}_\mathrm{R}.$$

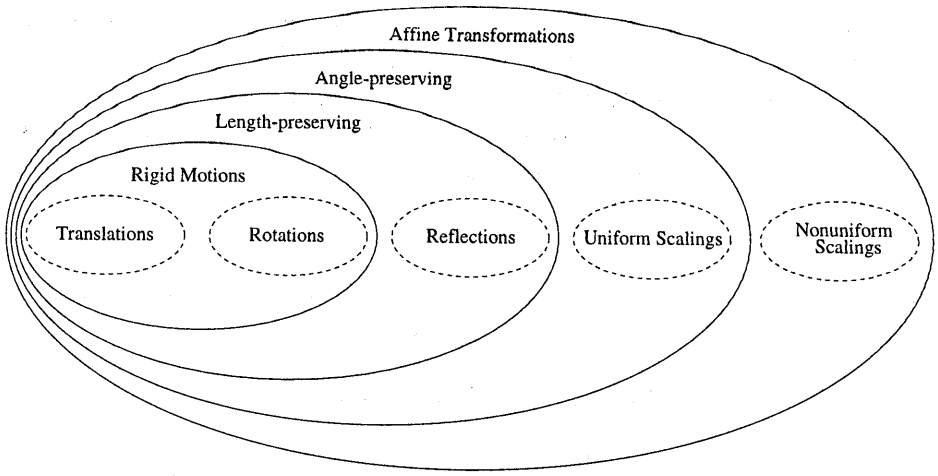


Figure A.2: The group of affine transformations. The dashed ellipses denote the sets of basic transformations, which are the components from which arbitrary affine transformations are built. Each group of transformations denoted by a solid ellipse is constituted by compositions of basic transformations from the sets inside the ellipse.

Here, the rotation and scaling components of an affine transformation can be found by singular value decomposition of the linear part. Figure A.2 shows a visual representation of the group of affine transformations.

For $\mathbf{n} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ and $\delta \in \mathbb{R}$, the **(hyper)plane** $H(\mathbf{n}, \delta)$ in $\mathbb{R}^n$ is a set of points defined by

$$H(\mathbf{n}, \delta) = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{n} \cdot \mathbf{x} + \delta = 0\}$$

The vector $\mathbf{n}$ is referred to as a **normal** of the hyperplane. For $\|\mathbf{n}\| = 1$, it can be shown that the distance from $H(\mathbf{n}, \delta)$ to a point $\mathbf{p}$ is $|\mathbf{n} \cdot \mathbf{p} + \delta|$. It is left as an exercise to the reader to show that a hyperplane is an affine set.

Let H' be the image of a hyperplane H under affine transformation $\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$. A normal $\mathbf{n}'$ and a scalar δ' such that

$$H' = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{n}' \cdot \mathbf{x} + \delta' = 0\}$$

are found as follows. For $\mathbf{x} \in H'$ we deduce

$$\begin{aligned} \mathbf{n} \cdot \mathbf{B}^{-1}(\mathbf{x} - \mathbf{c}) + \delta = 0 &\equiv \mathbf{n}^{\mathrm{T}}\mathbf{B}^{-1}(\mathbf{x} - \mathbf{c}) + \delta = 0 \\ &\equiv ((\mathbf{B}^{-1})^{\mathrm{T}}\mathbf{n})^{\mathrm{T}}(\mathbf{x} - \mathbf{c}) + \delta = 0 \\ &\equiv (\mathbf{B}^{-1})^{\mathrm{T}}\mathbf{n} \cdot (\mathbf{x} - \mathbf{c}) + \delta = 0. \end{aligned}$$

We see that $\mathbf{n}' = (\mathbf{B}^{-1})^{\mathrm{T}}\mathbf{n}$ and $\delta' = \delta - \mathbf{n}' \cdot \mathbf{c}$ yields $\mathbf{n}' \cdot \mathbf{x} + \delta' = 0$. If $\mathbf{T}$ is length-preserving then $\mathbf{B}$ is orthogonal, and thus $(\mathbf{B}^{-1})^{\mathrm{T}} = \mathbf{B}$, in which case we may transform a normal in the same way as a vector that is the difference of two points.

The positive closed **halfspace** defined by a hyperplane $H(\mathbf{n}, \delta)$ is defined as

$$H^{+}(\mathbf{n}, \delta) = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{n} \cdot \mathbf{x} + \delta \geq 0\}$$

The positive open halfspace defined by a hyperplane $H(\mathbf{n}, \delta)$ is defined as

$$H^{\oplus}(\mathbf{n}, \delta) = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{n} \cdot \mathbf{x} + \delta > 0\}$$

The negative closed and positive open halfspaces are defined similarly.

A.6 Three-dimensional Space

Here, we will discuss some concepts that apply to three-dimensional space only. By convention, the world coordinate system in $\mathbb{R}^3$ is a right-handed Cartesian system. A coordinate system relative to the world coordinate system is called **right-handed** if the matrix $[\mathbf{b}_1\ \mathbf{b}_2\ \mathbf{b}_3]$ has a positive determinant, where $\mathbf{b}_i$ are the basis vectors in world coordinates.

The **cross product** of two vectors $\mathbf{v}$ and $\mathbf{w}$, denoted by $\mathbf{v} \times \mathbf{w}$, is a vector determined by the following rules

1. orthogonal: $(\mathbf{v} \times \mathbf{w}) \perp \mathbf{v}$ and $(\mathbf{v} \times \mathbf{w}) \perp \mathbf{w}$.
2. positively oriented: $\det[\mathbf{v}\ \mathbf{w}\ \mathbf{v} \times \mathbf{w}] > 0$ for $\mathbf{v}$, $\mathbf{w}$ linearly independent.
3. $\|\mathbf{v} \times \mathbf{w}\| = \|\mathbf{v}\|\|\mathbf{w}\| \sin(\alpha)$, where α is the angle between $\mathbf{v}$ and $\mathbf{w}$.

It can be shown that, for vectors relative to an orthonormal basis, the cross product is given by

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} \times \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} \alpha_2\beta_3 - \alpha_3\beta_2 \\ \alpha_3\beta_1 - \alpha_1\beta_3 \\ \alpha_1\beta_2 - \alpha_2\beta_1 \end{bmatrix}.$$

Appendix B

User's Guide to SOLID 2.0

B.1 Introduction

SOLID is a library for collision detection of three-dimensional objects undergoing rigid motion and deformation. SOLID is designed to be used in interactive 3D graphics applications, and is especially suited for collision detection of objects and worlds described in VRML. Some of its features are:

- Object shapes are represented by primitive shapes (box, cone, cylinder, sphere), and complexes of polytopes (line segments, convex polygons, convex polyhedra). A single shape can be used to instantiate multiple objects.
- Motion is specified by translations, rotations, and nonuniform scalings of the local coordinate system of each moving object. These changes can be given absolute or relative to the previous frame. The local coordinate system can also be set according to an array of sixteen floats or doubles representing a 4x4 column-major matrix of an affine transformation, as used in OpenGL.
- Deformations of complex shapes can be specified using client-defined vertex arrays.
- Collision response is defined by the client by means of call-back functions. Response may be defined per object pair, for all pairs containing a specific object, and as default for all pairs of objects.
- Response call-backs can use collision data describing the configuration of a pair of colliding objects. As collision data can be used: a

point common to both objects, or the closest point pair of the objects from the previous frame. The latter response type can be used for approximating the collision plane in physics-based simulations.

- Frame coherence is exploited by maintaining a set of pairs of proximate objects (incremental sweep and prune of axis-aligned bounding boxes), and caching separating axes for these pairs. This feature is optional and may be turned on/off at any time during a simulation.

SOLID is written in standard C++ and relies heavily on the STL. The library has a standard C API and can be linked to both C and C++ applications.

B.2 Building Shapes

The commands for creating and destroying shapes are

```
DtShapeRef dtBox(DtScalar x, DtScalar y, DtScalar z);
DtShapeRef dtCone(DtScalar radius, DtScalar height);
DtShapeRef dtCylinder(DtScalar radius,
                      DtScalar height);
DtShapeRef dtSphere(DtScalar radius);
DtShapeRef dtNewComplexShape();

void dtDeleteShape(DtShapeRef shape);
```

Shapes are referred to by values of `DtShapeRef`. Currently, the type `DtScalar` is defined as `double`. The command `dtBox` creates a rectangular parallelepiped centered at the origin and aligned with the axes of the shape's local coordinate system. The parameters specify its extent along the respective coordinate axes. The commands `dtCone` and `dtCylinder` create respectively a cone and a cylinder centered at the origin and whose central axis is aligned with the y-axis of the local coordinate system. The cone's apex is at `y = height / 2`. The command `dtSphere` creates a sphere centered at the origin of the local coordinate system.

Other shape types based on point data, such as polygon soups, simplicial complexes, (compositions of) convex polyhedra, are created by the `dtNewComplexShape` command. For constructing complex shapes the following commands are used:

```
DtShapeRef dtNewComplexShape();
void dtEndComplexShape();
```

```
void dtBegin(DtPolyType type);
void dtEnd();
void dtVertex(DtScalar x, DtScalar y, DtScalar z);

void dtVertexBase(const void *base);
void dtVertexIndex(DtIndex index);
void dtVertexIndices(DtPolyType type, DtCount count,
                     const DtIndex *indices);
void dtVertexRange(DtPolyType type, DtIndex first,
                   DtCount count);
```

A complex shape is composed of d-dimensional polytopes, where d is at most 3. A d-polytope can be a simplex (point, line segment, triangle, tetrahedron), a convex polygon, or a convex polyhedron. The type of d-polytope is specified by a value of `DtPolyType`, defined as

```
typedef enum DtPolyType {
  DT_SIMPLEX,
  DT_POLYGON,
  DT_POLYHEDRON
} DtPolyType;
```

A d-polytope is specified by enumerating its vertices. This can be done in two ways. In the first way, the vertices are specified by value, using the `dtVertex` command. The following example shows how the facets of a pyramid are specified.

```
DtShapeRef pyramid = dtNewComplexShape();

dtBegin(DT_SIMPLEX);
dtVertex(1.0, 0.0, 1.0);
dtVertex(1.0, 0.0, -1.0);
dtVertex(-1.0, 0.0, -1.0);
dtVertex(-1.0, 0.0, 1.0);
dtEnd();

dtBegin(DT_SIMPLEX);
dtVertex(1.0, 0.0, 1.0);
dtVertex(1.0, 0.0, -1.0);
dtVertex(0.0, 1.27, 0.0);
dtEnd();
```

```
...

dtEndComplexShape();
```

In the second method, the vertices are stored in an array, and are referred to by indices. For each complex shape, we can specify a single array. A vertex array is specified by the command `dtVertexBase`, which takes the address of the first element in the array, referred to as the base of the array, as argument. The command `dtVertexIndex` is used for specifying vertices. See the following example:

```
DtScalar vertices[5 * 3] = {
  1.0, 0.0, 1.0,
  1.0, 0.0, -1.0,
  -1.0, 0.0, -1.0,
  -1.0, 0.0, 1.0,
  0.0, 1.27, 0.0
};

DtShapeRef pyramid = dtNewComplexShape();
dtVertexBase(vertices);

dtBegin(DT_SIMPLEX);
dtVertexIndex(0);
dtVertexIndex(1);
dtVertexIndex(2);
dtVertexIndex(3);
dtEnd();

dtBegin(DT_SIMPLEX);
dtVertexIndex(0);
dtVertexIndex(1);
dtVertexIndex(4);
dtEnd();

...

dtEndComplexShape();
```

Alternatively, the indices can be placed into an array and specified using the command `dtVertexIndices`, as in the following example:

```
DtScalar vertices[5 * 3] = {
  1.0, 0.0, 1.0,
  1.0, 0.0, -1.0,
  -1.0, 0.0, -1.0,
  -1.0, 0.0, 1.0,
  0.0, 1.27, 0.0
};

DtIndex facet0[4] = { 0, 1, 2, 3 };
DtIndex facet1[3] = { 0, 1, 4 };

...

DtShapeRef pyramid = dtNewComplexShape();
dtVertexBase(vertices);

dtVertexIndices(DT_SIMPLEX, 4, facet0);
dtVertexIndices(DT_SIMPLEX, 3, facet1);

...

dtEndComplexShape();
```

Finally, a polytope can be specified from a range of vertices using the command `dtVertexRange`. The range is specified by the first index and the number of vertices. In the following example a pyramid is constructed as a convex polyhedron, which is the convex hull of the vertices in the array.

```
DtScalar vertices[5 * 3] = {
  1.0, 0.0, 1.0,
  1.0, 0.0, -1.0,
  -1.0, 0.0, -1.0,
  -1.0, 0.0, 1.0,
  0.0, 1.27, 0.0
};

DtShapeRef pyramid = dtNewComplexShape();
dtVertexBase(vertices);
dtVertexRange(DT_POLYHEDRON, 0, 5);
dtEndComplexShape();
```

Note that the vertices of a simplex need not be affinely independent, and the vertices specifying a convex polyhedron need not be extreme ver-

tices of the convex hull. However, in order to construct a proper convex polygon, the vertices should be approximately coplanar and specified in the order as they appear on the boundary.

B.3 Creating and Moving Objects

An object is an instance of a shape. The commands for creating, moving and deleting objects are

```
void dtCreateObject(DtObjectRef object,
                    DtShapeRef shape);
void dtDeleteObject(DtObjectRef object);
void dtSelectObject(DtObjectRef object);

void dtLoadIdentity();

void dtLoadMatrixf(const float *m);
void dtLoadMatrixd(const double *m);

void dtMultMatrixf(const float *m);
void dtMultMatrixd(const double *m);

void dtTranslate(DtScalar x, DtScalar y, DtScalar z);

void dtRotate(DtScalar x, DtScalar y, DtScalar z,
              DtScalar w);

void dtScale(DtScalar x, DtScalar y, DtScalar z);
```

An object is referred to by a `DtObjectRef`, which is defined as `void *`. Any pointer type may be used to refer to an object. In general, a pointer to a structure in the client application associated with the collision object should be used.

An object's motion is specified by changing the placement of the local coordinate system of the shape. Initially, the local coordinate system of an object coincides with the world coordinate system.

The *current object* is the last created or selected object. The placement of the current object is changed, either by translations, rotations and nonuniform scalings, or by using an OpenGL 4x4 column-major matrix representing an affine transformation. Placements are specified absolute or rel-

ative to the previous placement. Rotations are specified using quaternions. Following example shows how a pair of objects are given absolute placements.

```
dtCreateObject(&object1, hammer);
dtCreateObject(&object2, nail);

dtSelectObject(&object1);
dtLoadIdentity();
dtTranslate(0, 1, 1);
dtRotate(0, 0, 1, 0);

dtSelectObject(&object2);
dtLoadIdentity();
dtTranslate(0, 1, 0);
dtRotate(0, 0, 0, 1);
```

B.3.1 Who's Afraid of Quaternions?

A quaternion is a four-dimensional vector. The set of quaternions of length one (points on a four-dimensional sphere) map to the set of orientations in three-dimensional space. Since in many applications an orientation defined by either a rotation axis and angle or by a triple of Euler angles is more convenient, the package includes code for quaternion operations. The code is found in a library of C++ classes for 3D affine transformations. The classes are found in the `include/3D` directory. All the code is inlined so you do not need to link a library in order to use the classes.

The quaternion class is found in the file `Quaternion.h`. The class has constructors and methods for setting a quaternion. For example

```
Quaternion q1(axis, angle);
Quaternion q2(yaw, pitch, roll);

...

q1.setRotation(axis, angle);
q2.setEuler(yaw, pitch, roll);

...

dtRotate(q1[X], q1[Y], q1[Z], q1[W]);
```

Also included is a static method `Quaternion::random()`, which returns a random orientation.

B.4 Response Handling

Collision response in SOLID is handled by means of callback functions. The callback functions have the type `DtResponse` defined by

```
typedef void (*DtResponse)(
  void *client_data,
  DtObjectRef object1,
  DtObjectRef object2,
  const DtCollData *coll_data)
```

Here, `client_data` is a pointer to an arbitrary structure in the client application, `object1` and `object2` are colliding objects, and `coll_data` is the response data computed by SOLID.

Currently, there are three types of response: *simple*, *smart* and *witnessed* response. For simple response the value of `coll_data` is `NULL`. For smart and witnessed response `coll_data` points to the following structure

```
typedef struct DtCollData {
  DtVector point1;
  DtVector point2;
  DtVector normal;
} DtCollData;
```

An object of this type represents a pair of points of the respective objects. The points `point1` and `point2` are given relative to the local coordinate system of their respective objects `object1` and `object2`. The `normal` field is used for smart response only.

For witnessed response, the points represent a witness of the collision. As a result of this the global coordinates of the witness points are equal. For smart response, the points represent the closest point pair of the objects at placements from the previous time frame. The `normal` field contains the difference of the global coordinates of the closest point pair, i.e., `normal = Global(point1) - Global(point2)`. We will discuss this type of response more thoroughly further on.

Response is defined as *default response* for all pairs of objects, as *object response* for all pairs containing a given object, or as *pair response* for a particular pair of objects. The commands for defining and undefining response are

```
void dtSetDefaultResponse(DtResponse response,
                          DtResponseType type,
                          void *client_data)
void dtClearDefaultResponse()

void dtSetObjectResponse(DtObjectRef obj,
                         DtResponse response,
                         DtResponseType type,
                         void *client_data)
void dtClearObjectResponse(DtObjectRef obj)
void dtResetObjectResponse(DtObjectRef obj)

void dtSetPairResponse(DtObjectRef obj1,
                       DtObjectRef obj2,
                       DtResponse response,
                       DtResponseType type,
                       void *client_data)
void dtClearPairResponse(DtObjectRef obj1,
                         DtObjectRef obj2)
void dtResetPairResponse(DtObjectRef obj1,
                         DtObjectRef obj2)
```

A response is defined by either a `Set` or a `Clear` command. The `Clear` command defines the response to be nil (no response).

Initially, the default response is nil and all pairs of objects have a default response. If for an object pair, one of the objects has an object response defined, then this response overrules the default response. A pair response overrules any object or default response. If for both objects there is an object response defined, then one of the responses is chosen. In this case, one of the responses may be forced to be chosen by defining it as a pair response.

A response is undefined, i.e., reset to a more general setting, by the `Reset` commands. The command `dtResetPairResponse` resets the response of a pair of objects to an object response, if one is defined for an object in the pair, or otherwise to the default response. The command `dtResetObjectResponse` resets the responses of the object pairs, for which no other object response or a pair response is defined, to the default response. Note that whenever an object is deleted, the object response and all pair responses that are set for this object are reset automatically.

The `DtResponseType` is defined by

```
typedef enum DtResponseType {
  DT_NO_RESPONSE,
  DT_SIMPLE_RESPONSE,
  DT_SMART_RESPONSE,
  DT_WITNESSED_RESPONSE
} DtResponseType
```

Setting the response type to `DT_NO_RESPONSE` is equivalent to clearing the response.

The response callback functions are executed for each colliding pair of objects by calling

```
int dtTest()
```

This function returns the number of callback functions that are executed.

B.4.1 Smart Response

For physics-based simulations it is often necessary to have a representation of the collision plane of a pair of colliding objects in order to compute the reaction forces. From a single configuration of two colliding objects it is hard to compute a collision plane, since there is no knowledge of how this configuration came about. Therefore, SOLID uses the configuration of the objects from the previous time frame for approximating the collision plane. If the objects were disjoint in the previous time frame, then the vector defined by the difference of the closest point pair of the objects is a good approximation of the collision plane's normal.

By selecting smart response for a pair of objects, the closest point pair and the normal from the previous time frame are computed. The points `point1` and `point2` are given in local coordinates and the `normal` relative to the global basis and pointing away from `object2`. In order to compute these values, the configuration of objects must be stored in each time frame. This is done by calling

```
void dtProceed();
```

Note that in order to guarantee that a nonzero normal can be found, the `dtProceed` command may only be called if all object pairs for which a smart response is defined, are disjoint! The common way of guarding this is by iteratively doing collision tests and changing the placements until the objects are disjoint. Note that it is possible and often necessary to call `dtTest` multiple times before calling `dtProceed`.

B.5 Deformable Models

SOLID handles deformations of complex shapes. In this context deformations are specified by changes of vertex positions. Complex shapes that are defined using a vertex array in the client application may be deformed by changing the array elements, or specifying a new array. SOLID is notified of a change of vertices by the command

```
void dtChangeVertexBase(DtShapeRef shape,
                        const void *base);
```

When using convex polygons or convex polyhedra as shape components, the client should warrant that the vertex changes do not violate the convexity and topology (planar graph embedding) of a component!

Note that in order to use smart response for deformable shapes, the change of vertices should be done by specifying a new array. The vertex array of the previous time frame should be kept intact, otherwise SOLID can not determine the configuration of objects of the previous time frame. This is best handled by applying a 'double buffering' technique. After a call to `dtProceed`, the new vertex positions are placed in the free buffer of a pair of vertex buffers, and `dtChangeVertexBase` is called using this buffer, after which the other buffer becomes the free buffer.

B.6 Caching

In computer animations there is usually a lot of frame coherence (objects move smoothly). In these cases, caching and reusing earlier computations will yield a considerable performance improvement. The *caching* option of SOLID enables an incremental sort on the set of objects, in order to reduce the number of pairwise intersection tests. Moreover, when the cashing option is on, data from earlier intersection tests is stored and used for faster determination of the intersection status of a pair of objects. Caching is enabled and disabled by

```
void dtEnableCaching()
void dtDisableCaching()
```

Caching may be enabled or disabled at any time during a simulation. This option is enabled by default.

Bibliography

[1] M. Abrash. *Zen of Graphics Programming*. The Coriolis Group, Scottsdale, AZ, 1996.

[2] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Proc. SIGGRAPH '89*, volume 23, pages 223–232, 1989.

[3] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In *Proc. SIGGRAPH '90*, volume 24, pages 19–28, 1990.

[4] D. Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Computer Science Department, Cornell University, 1992. Technical Report 92-1275.

[5] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hull. *ACM Transactions on Mathematical Software*, 22:469–483, 1996.

[6] B. Barenbrug. *Designing a Class Library for Interactive Simulation of Rigid Body Dynamics*. PhD thesis, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, Dec. 1998.

[7] B. G. Baumgart. A polyhedron representation for computer vision. In *Proc. AFIPS National Computer Conference*, volume 44, pages 589–596, 1975.

[8] G. Bell, R. Carey, and C. Marrin. VRML97: The Virtual Reality Modeling Language. http://www.vrml.org/Specifications/VRML97, 1997.

[9] J. F. Blinn. A trip down the graphics pipeline: Line clipping. *IEEE Computer Graphics and Applications*, 11(1):98–105, 1991.

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

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

[12] S. Cameron. Collision detection by four-dimensional intersection testing. *IEEE Transactions on Robotics and Automation*, 6(3):291–302, 1990.

[13] S. Cameron. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3112–3117, 1997.

[14] S. A. Cameron and R. K. Culley. Determining the minimum translational distance between convex polyhedra. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 591–596, 1986.

[15] J. Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(2):200–209, 1986.

[16] F. Cazals and C. Puech. Bucket-like space partitioning data structures with applications to ray-tracing. In *Proc. 13th Annual ACM Symposium on Computational Geometry*, pages 11–20, 1997.

[17] B. Chazelle and D. P. Dobkin. Detection is easier than computation. In *Proc. 12th Annual ACM Symposium on Theory of Computing*, pages 146–153, 1980.

[18] K. Chung. An efficient collision detection algorithm for polytopes in virtual environments. Master's thesis, University of Hong Kong, Sept. 1996.

[19] K. Chung. Q-COLLIDE: Quick collision detection library. http://www.csis.hku.hk/~tlchung/collision_library.html, 1996. web page.

[20] K. Chung and W. Wang. Quick collision detection of polytopes in virtual environments. In *Proc. ACM Symposium on Virtual Reality Software and Technology*, pages 125–131, 1996.

[21] J. Cohen, M. C. Lin, D. Manocha, B. Mirtich, M. K. Ponamgi, and J. Canny. I-COLLIDE: Interactive and exact collision detection library. http://www.cs.unc.edu/~geom/I_COLLIDE.html, 1996. software library.

[22] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proc. ACM Symposium on Interactive 3D Graphics*, pages 189–196, 1995.

[23] H. S. M. Coxeter. *Introduction to Geometry*. Wiley, New York, NY, 2nd edition, 1989.

[24] R. K. Culley and K. G. Kempf. A collision detection algorithm based on velocity and distance bounds. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1064–1069, 1986.

[25] M. de Berg. Linear size binary space partitions for fat objects. In *Proc. 3rd Annu. European Sympos. Algorithms*, volume 979 of *Lecture Notes in Computer Science*, pages 252–263. Springer-Verlag, 1995.

[26] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoretical Computer Science*, 27:241–253, 1983.

[27] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *Journal of Algorithms*, 6:381–392, 1985.

[28] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra — a unified approach. In *Proc. 17th Int. Coll. Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 400–413. Springer-Verlag, 1990.

[29] V. J. Duvanenko, W. E. Robbins, and R. S. Gyurcsik. Improving line segment clipping. *Dr. Dobb's Journal*, pages 36–45, July 1990.

[30] V. J. Duvanenko, W. E. Robbins, and R. S. Gyurcsik. Line-segment clipping revisited. *Dr. Dobb's Journal*, pages 107–110, Jan. 1996.

[31] M. E. Dyer. Linear time algorithms for two- and three-variable linear programs. *SIAM Journal on Computing*, 13:31–45, 1984.

[32] J. Fenlason and R. Stallman. *GNU gprof: The GNU Profiler*. Free Software Foundation, 1992.

[33] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 2nd edition, 1990.

[34] A. U. Frank and R. Barrera. The Fieldtree: A data structure for geographic information systems. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, editors, *Proc. 1st Symposium SSD on Design and Implementation af Large Spatial Databases*, volume 409 of *Lecture Notes in Computing Science*, pages 29–44. Springer-Verlag, July 1990.

[35] Free Software Foundation. *GNU Library General Public License*, 1991. http://www.fsf.org/copyleft/lgpl.html.

[36] K. Y. Fung, T. M. Nicholl, R. E. Tarjan, and C. J. Van Wyk. Simplified linear-time Jordan sorting and polygon clipping. *Information Processing Letters*, 35:85–92, 1990.

[37] M. A. Ganter and B. P. Isarankura. Dynamic collision detection using space partitioning. *Journal of Mechanical Design*, 115:150–155, Mar. 93.

[38] A. Garcia-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 14:36–43, May 1994.

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

[40] E. G. Gilbert, D. W. Johnson, and S. S. 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.

[41] A. S. Glassner. *An Introduction to Ray Tracing.* Academic Press, 1989.

[42] A. S. Glassner. Clipping a concave polygon. In A. W. Paeth, editor, *Graphics Gems V*, pages 50–54. Academic Press, Boston, MA, 1995.

[43] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[44] S. Gottschalk. RAPID: Robust and accurate polygon interference detection system. http://www.cs.unc.edu/~geom/OBB/OBBT.html, 1996. software library.

[45] S. Gottschalk. Separating axis theorem. Technical Report TR96-024, Dept. of Computer Science, UNC Chapel Hill, 1996.

[46] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proc. SIGGRAPH '96*, pages 171–180, 1996.

[47] D. Green and D. Hatch. Fast polygon-cube intersection testing. In A. W. Paeth, editor, *Graphics Gems V*, pages 375–379. Academic Press, Boston, MA, 1995.

[48] B. Grünbaum. *Convex Polytopes*. Wiley, New York, NY, 1967.

[49] J. K. Hahn. Realistic animation of rigid bodies. In *Proc. SIGGRAPH '88*, volume 22, pages 299–308, 1988.

[50] E. Haines. Point in polygon strategies. In P. Heckbert, editor, *Graphics Gems IV*, pages 24–46. Academic Press, Boston, MA, 1994.

[51] S. Hawking. *A Brief History of Time*. Bantam Books, London, UK, 1988.

[52] M. Held. ERIT – A collection of efficient and reliable intersection tests. *Journal of Graphics Tools*, 1998. to appear.

[53] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time using level-linked search trees. *Information and Control*, 68:170–184, 1986.

[54] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. Efficient detection of intersections among spheres. *International Journal of Robotics Research*, 2(4):77–80, 1983.

[55] H. Hoppe. Progressive meshes. In *Proc. SIGGRAPH '96*, pages 99–108, 1996.

[56] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[57] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annual ACM Symposium on Computational Geometry*, pages 146–154, 1998.

[58] D.-J. Kim, L. J. Guibas, and S.-Y. Shin. Fast collision detection among multiple moving spheres. In *Proc. Computer Animation '97*, pages 1–7, 1997.

[59] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[60] Y.-D. Liang and B. A. Barsky. A new concept and method for line clipping. *ACM Transactions on Graphics*, 3:1–22, 1984.

[61] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance computation. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1008–1014, 1991.

[62] D. G. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1984.

[63] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal on Computing*, 12:759–776, 1983.

[64] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1984.

[65] S. Meyers. *More Effective C++*. Professional Computing Series. Addison-Wesley, Reading, MA, 1996.

[66] B. Mirtich. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, 1997.

[67] T. Möller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.

[68] M. Moore and J. Willhelms. Collision detection and response for computer animation. In *Proc. SIGGRAPH '88*, volume 22, pages 289–298, 1988.

[69] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

[70] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Professional Computing Series. Addison-Wesley, Reading MA, 1996.

[71] J. Nagle. GJK collision detection algorithm wanted. posted on the *comp.graphics.algorithms* newsgroup, Apr. 1998.

[72] A. Narkhede and D. Manocha. Fast polygon triangulation based on Seidel's algorithm. In A. W. Paeth, editor, *Graphics Gems V*, pages 394–397. Academic Press, 1995.

[73] B. Naylor, J. A. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. In *Proc. SIGGRAPH '90*, volume 24, pages 115–124, 1990.

[74] B. F. Naylor. Interactive solid geometry via partitioning trees. In *Proc. Graphics Interface '92*, pages 11–18, 1992.

[75] C. J. Ong and E. G. Gilbert. The Gilbert-Johnson-Keerthi distance algorithm: A fast version for incremental motions. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1183–1189, 1997.

[76] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York, NY, 1987.

[77] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, NY, 2nd edition, 1998.

[78] I. J. Palmer and R. L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105–116, 1995.

[79] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry*, 5:485–503, 1990.

[80] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[81] E. Schömer and C. Thiel. Efficient collision detection for moving polyhedra. In *Proc. 11th Annual ACM Symposium on Computational Geometry*, pages 51–60, 1995.

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

[83] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 2nd edition, 1988.

[84] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete and Computational Geometry*, 6:423–434, 1991.

[85] K. Shoemake. Animating rotations with quaternion curves. In *Proc. SIGGRAPH '85*, volume 19, pages 245–254, July 1985.

[86] K. Shoemake. Uniform random rotations. In D. Kirk, editor, *Graphics Gems III*, pages 124–132. Academic Press, Boston, MA, 1992.

[87] H. W. Six and D. Wood. Counting and reporting intersections of d-ranges. *IEEE Transactions on Computers*, C-31:181–187, 1982.

[88] A. Smith, Y. Kitamura, H. Takemura, and F. Kishino. A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. In *Proc. IEEE Virtual Reality Annual International Symposium*, pages 136–145, 1995.

[89] B. J. A. Snepvangers. Collision detection of convex primitive shapes. Master's thesis, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, Dec. 1997.

[90] D. Stoyan, W. S. Kendall, and J. Mecke. *Stochastic Geometry and Its Applications*. Wiley, Chichester, UK, 1987.

[91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

[92] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17:32–42, 1974.

[93] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, Mar. 1974.

[94] F. Tampieri. Newell's method for computing the plane equation of a polygon. In D. Kirk, editor, *Graphics Gems III*, pages 231–232. Academic Press, Boston, MA, 1992.

[95] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *Proc. SIGGRAPH '87*, volume 21, pages 153–162, 1987.

[96] F. Thomas and C. Torras. Interference detection between non-convex polyhedra revisited with a practical aim. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 587–594, 1994.

[97] E. P. C. F. M. van der Laak. Adaptive octrees for balancing the load of a parallel simulation of a cluster-cluster aggregation. Master's thesis, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, Aug. 1996.

[98] D. Voorhies. Triangle-cube intersection. In D. Kirk, editor, *Graphics Gems III*, pages 236–239. Academic Press, Boston, MA, 1992.

[99] R. Webb and M. Gigante. Using dynamic bounding volume hierarchies to improve efficiency of rigid body simulations. In T. L. Kunii, editor, *Visual Computing (Proc. CG International '92)*, pages 825–842. Springer-Verlag, 1992.

[100] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, Jan. 1994.

[101] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.

[102] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 2nd edition, 1997.

[103] X. Wu. A linear-time simple bounding volume algorithm. In D. Kirk, editor, *Graphics Gems III*, pages 301–306. Academic Press, Boston, MA, 1992.

[104] G. Zachmann. *Exact and Fast Collision Detection*. PhD thesis, Dept. of Computer Science, TU Darmstadt, 1994.

[105] G. Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. IEEE Virtual Reality Annual International Symposium*, pages 90–97, 1998.

[106] G. Zachmann and W. Felger. The BoxTree: Enabling real-time and exact collision detection of arbitrary polyhedra. In *Proc. Workshop on Simulation and Interaction in Virtual Environments*, pages 104–113, 1995.

[107] M. J. Zyda, D. R. Pratt, W. D. Osborne, and J. G. Monahan. NPSNET: Real-time collision detection and response. *Journal of Visualization and Computer Animation*, 4(1):13–24, 1993.

Index

Samenvatting

Met de huidige interactieve computeranimatie-systemen kunnen complexe driedimensionale omgevingen gesimuleerd worden. Zo'n gesimuleerde omgeving bevat in veel gevallen bewegende elementen, bijvoorbeeld in 3D computerspellen en simulatoren. We duiden dergelijke toepassingen ook wel aan met de term *virtuele werkelijkheid*.

Zeer bepalend voor het realiteitsgehalte van een gesimuleerde omgeving is de beperking dat twee materiële objecten niet tegelijkertijd hetzelfde punt in de ruimte kunnen innemen. Geometrische objecten die gerepresenteerd worden met behulp van computermodellen zijn over het algemeen niet aan deze beperking onderworpen. Configuraties van elkaar doorsnijdende paren van objecten, *botsingen* genaamd, kunnen voorkomen en moeten opgelost worden door het animatie-systeem.

De eerste taak hierin is het detecteren van botsingen. Verder moet het animatie-systeem vaak gegevens hebben over de configuratie van de botsende objecten om de botsingen op te kunnen lossen. Bijvoorbeeld, voor het berekenen van de elastische reactiekrachten van een paar botsende objecten hebben we een botsingsvlak en een botsingspunt nodig. We noemen deze gegevens *responsgegevens*.

In dit proefschrift behandelen we methoden voor het detecteren van botsingen tussen bewegende driedimensionale objecten. We beschouwen botsingsdetectie-methoden voor objecten gerepresenteerd door modellen die bestaan uit geometrische primitieven, zoals polygonen, convexe polyhedra, bollen, kegels, en cilinders. Daarnaast beschrijven we methoden voor het bepalen van responsgegevens voor deze object-typen.

De belangrijkste bijdragen van dit proefschrift zijn:

- Een verbeterde algoritme voor het detecteren van botsingen tussen convexe objecten. Deze algoritme, ISA-GJK genaamd, is een verbetering van de Gilbert-Johnson-Keerthi algoritme, een algoritme voor het berekenen van de afstand tussen twee convexe objecten. De verbeteringen hebben betrekking op de performance, de robuustheid en de toepasbaarheid.

- Een datastructuur, AABB-tree genaamd, voor het versnellen van botsingsdetectie tussen modellen die uit een groot aantal primitieven bestaan. Het sterkste punt van de AABB-tree is het feit dat hij snelle botsingsdetectie tussen deformeerbare objecten mogelijk maakt.

- Een software-bibliotheek voor het detecteren van botsingen tussen driedimensionale objecten, genaamd SOLID. Enkele innovatieve kenmerken van SOLID zijn: (a) het ondersteunen van een mix van primitieve typen, (b) het ondersteunen van complexe deformeerbare vormen, en (c) het bepalen van een botsingsvlak en een botsingspunt als responsgegevens voor fysische simulaties.

Dankwoord

Het schrijven van een proefschrift is een solistische bezigheid, maar zonder de ondersteuning van anderen is het vrijwel ondoenlijk. Op deze plaats wil ik de mensen bedanken die hebben bijgedragen aan dit werk.

Allereerst bedank ik de leden van de leescommissie: Dieter Hammer, Kees van Overveld, Mark Overmars, Frans Peters en Stephen Cameron voor het becommentariëren van de aangeleverde manuscripten.

Verder bedank ik Huub van de Wetering voor zijn belangstelling en adviezen, Wieger Wesselink voor zijn wiskundige inbreng, Elisabeth Melby en Alex de Cock voor het programmeren van demo's en benchmarks, de afstudeerders en stagiairs: Barry Snepvangers, Marcel Marsman, Laurens van der Laar en Bart van den Broek, en de medewerkers van de bibliotheek en het bureau computerfaciliteiten van de faculteit Wiskunde en Informatica.

Ook bedank ik mijn collega's, in het bijzonder Richard Kelleners, Paul van Gorp en Maarten Bodlaender, voor de gezellige tijd die ik heb gehad op de universiteit.

Tenslotte gaat mijn dank bovenal uit naar Marja, mijn levensgezellin. Marja, dank je wel voor jouw liefde, geduld, motivatie, inzichten, en secretariële en mentale ondersteuning.

Curriculum Vitae

Gino van den Bergen is geboren op 21 juni 1969 te Hulst. Na het behalen van het diploma atheneum-bèta aan de R. K. Jansenius Scholengemeenschap te Hulst begon hij in 1987 aan een studie informatica aan de Technische Universiteit Eindhoven. In december 1992 behaalde hij 'met lof' het doctoraal examen in de technische informatica. Zijn afstudeerscriptie had als onderwerp, het bijhouden van de contour van de vereniging van een dynamische collectie assen-parallelle rechthoeken.

Na zijn studie is hij nog enkele maanden in dienst geweest van de Technische Universiteit Eindhoven als toegevoegd onderzoeker voor het schrijven van een publicatie naar aanleiding van zijn afstudeerwerk. Vervolgens heeft hij de militaire dienstplicht vervult, waarna hij in 1994 opnieuw in dienst kwam van de Technische Universiteit Eindhoven, deze keer als assistent-in-opleiding bij de vakgroep Informatica. Daar heeft hij tot 1999 onderzoek verricht binnen de groep Computer Graphics, wat uiteindelijk resulteerde in dit proefschrift. Sinds januari 1999 werkt hij als software-engineer bij VDO Car Communication aan het CARiN navigatie-systeem.

STELLINGEN

bij het proefschrift

Collision Detection in Interactive 3D Computer Animation

van

GINO VAN DEN BERGEN

—1—

De algoritme in [1] voor het testen van intersecties tussen driehoeken is zonder meer toepasbaar op convexe polygonen in het algemeen.

[1] T. Möller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.

—2—

Het bewijs van convergentie van de iteratieve methode voor het vinden van een separating axis van een paar polytopen, beschreven in [1], is niet correct. De algoritme in [1] gebruikt naast de iteratiestap uit dit bewijs nog een tweede iteratiestap die wel eindiging garandeert voor polytopen. Deze tweede iteratiestap is van cruciaal belang voor de eindiging van de algoritme, en is niet slechts nodig vanwege numerieke onnauwkeurigheden, zoals beweerd in [1].

[1] K. Chung and W. Wang. Quick collision detection of polytopes in virtual environments. In *Proc. ACM Symposium on Virtual Reality Software and Technology* pages 125–131, 1996.

—3—

In tegenstelling tot het probleem van het vinden van een gemeenschappelijk punt van een paar polytopen, kan het probleem van het vinden van een *separating plane* tussen twee polytopen, zoals gedefinieerd in dit proefschrift, niet als een lineair-programmeringsprobleem uitgedrukt worden.

—4—

De bewering in [1] dat de algoritme in [2], voor het rapporteren van alle elkaar overlappende paren in een collectie d-dimensionale hyperrechthoeken, een lagere tijdscomplexiteit heeft dan de algoritme in [3] is niet juist. Gegeven een collectie van n hyperrechthoeken, kost het met de algoritme in [2] namelijk $O(n \log^{2d-3} n + k)$ tijd om alle k elkaar overlappende paren te rapporteren, terwijl met de algoritme in [3] hiervoor $O(n \log^{d-1} n + k)$ tijd nodig is.

[1] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[2] H. Edelsbrunner. A new approach to rectangle intersections, Part II. *Intern. J. Computer Math.*, 13:221–229, 1983.

[3] H. W. Six and D. Wood. Counting and reporting intersections of d-ranges. *IEEE Transactions on Computers*, C-31:181–187, 1982.

—5—

De kracht van een softwarebibliotheek ligt in de balans tussen functionaliteit en flexibiliteit. De functies die een softwarebibliotheek aanbiedt mogen de gebruiker zo min mogelijk beperken in de keuze van zijn datarepresentaties. Het succes van *OpenGL* en de *Standard Template Library* is hiermee te verklaren.

—6—

Het template-mechanisme van de programmeertaal C++ bevordert in grotere mate het hergebruik van code dan het inheritance-mechanisme.

—7—

De *methode van programmeren* van E. W. Dijkstra en W. H. J. Feijen [1] had een bredere belangstelling genoten indien deze zich aan de gangbare wiskundige, logische en algoritmische notaties hadden gehouden.

[1] E. W. Dijkstra en W. H. J. Feijen. *Een methode van programmeren*. Academic Service, Den Haag, 1984.

—8—

In een wereld waarin het software-aanbod steeds meer in handen komt van enkele grote bedrijven, is het de maatschappelijke taak van de universiteiten om tegenwicht te bieden aan de invloed van deze bedrijven door naast publicaties, software-technologie in de vorm van vrije software publiek beschikbaar te stellen. Met *vrije software* wordt hier bedoeld: software waarvan de broncode en de documentatie vrij beschikbaar is zodat een breed publiek er kennis van kan nemen en gebruik van kan maken.

—9—

In de praktijk blijkt vaak dat het eisenpakket dat aan een ontwerp wordt gesteld achteraf moet worden bijgesteld. Men kan in zo'n geval spreken van *reversed requirements engineering*.

—10—

De theorie dat de meest elementaire objecten in het universum snaren zijn, zoals recentelijk aangenomen door theoretische natuurkundigen, is al veel langer gemeengoed onder gitaristen.