Hierarchical Representation of 3D Polyhedra 2

2.1 Introduction

Collision detection is undoubtedly the most time-consuming step in a dynamic-simulation engine. In theory, as the simulation evolves, every object needs to be checked for collisions against all other objects in the simulation. Whenever a collision is detected, the simulation engine needs to backtrack in time to the instant before the collision, and determine the collision point and collision normal from the relative geometric displacement of the colliding objects.

Usually, collisions are checked by looking for geometric intersections between the objects. When objects are given by their boundary representations, this check can be done by looking for geometric intersections between the primitives of each object, that is, between the polygonal faces defining the boundary of each object. Clearly, checking for collisions between objects this way is a laborious task, and the use of intermediate representations to speed collision checking is critical to achieve real-time performance, especially for simulations involving several thousand objects, each described by several hundred primitives.

In this chapter, we study the use of hierarchical representations to speed the collision-detection phase. Our aim is to compute in a preprocessing stage the hierarchical volumetric decomposition of each object with respect to its local-coordinate frame. This usually consists of a tree hierarchy of bounding volumes where the top-most bounding volume bounds the entire object, the intermediate nodes of the tree bound sub-parts of the volume bounded by their parent, and the leaf nodes of the tree bound one or more primitives that lie inside the bounding volume of their parent. Collision checks are then carried out using the objects' hierarchical representations to quickly determine that the objects do not intersect (i.e., are not colliding), or to reduce the number of pair-wise primitive intersection tests needed to check for collision. For example, if the top-most bounding volumes of each object do not intersect, then we can safely conclude that the objects are not colliding. However, if the top-most bounding volumes do intersect, then we have to move down one level in the tree hierarchy to check whether their children intersect. If not, then the objects are not colliding. Otherwise, we move down one more level in the tree hierarchy to the

M.G. Coutinho, *Guide to Dynamic Simulations of Rigid Bodies and Particle Systems*,
Simulation Foundations, Methods and Applications,
DOI 10.1007/978-1-4471-4417-5_2, © Springer-Verlag London 2013

children of the intersecting parents. This process continues until we either reach the leaf nodes of the trees, or detect that the objects do not intersect. Should we reach the leaf nodes of the trees, the collision check proceeds by computing the pair-wise primitive intersections of the primitives bounded by each intersecting leaf.

In practice, there are two important points to consider when using hierarchical representations in a simulation engine. The first is that the intersection test of the bounding volumes must be considerably faster than the intersection test of the primitives. Otherwise, the collision check will take longer using the hierarchical representation than using the original objects' boundary representations. Therefore, our choice of bounding volumes is restricted to simple geometric shapes such as boxes and spheres, which can be quickly tested for intersection against each other. The primitives can also be restricted to convex polygons, or even triangles, to further speed the primitive–primitive intersection tests.

The second point addresses how the hierarchical representation is updated as the object translates and rotates with respect to the world-coordinate frame. As mentioned before, the hierarchical decomposition is computed with respect to the object's local-coordinate frame. However, all intersection tests should be carried out with respect to the world-coordinate frame, thus requiring a coordinate transformation from the object's local-coordinate frame to its position and orientation in the world-coordinate frame. One solution to this problem would be to transform the entire tree hierarchy of all objects to the world-coordinate frame, just before testing for intersection. The drawback of so doing is the substantial waste of time transforming entire tree hierarchies that have only their top-most, or even some of their internal nodes checked for intersection. All other internal nodes that were transformed but not used in the intersection tests were unnecessary transformed to the world-coordinate frame, and the time spent applying the transformation could have been saved. The idea is then to transform only what is absolutely necessary. We can do this as follows.

The simulation engine represents each object in the world-coordinate frame by its top-most bounding volume only, and keeps the entire tree hierarchy, as well as the object's boundary representation, in the object's local-coordinate frame. At each time step, *only* the top-most bounding volume of each object is moved in the world-coordinate frame. The movement consists of updating the position and orientation of the object according to the numerical method being used,[1] applying it to the top-most bounding volume of the object.

The collision-detection phase then checks for geometric intersections between the top-most bounding volumes that can potentially collide. Potentially colliding objects are determined from the world-cell decomposition structure, as explained in Sect. 2.4, or from the world-tree hierarchy in the case of continuous collision detection, as explained in Sect. 2.4.3. Whenever the top-most bounding volumes intersect, the simulation engine transforms only their next-level children from their local-coordinate frame to the world-coordinate frame. If their next-level children do not intersect, the objects are not colliding and no further transformations are

[1] This is discussed in detail in Appendix B (Chap. 7).

required. Otherwise, the simulation engine keeps transforming only the next-level children of the intersecting bounding volumes until it concludes that the objects are not intersecting, or have intersecting leaf nodes. In that case, each primitive associated with the intersecting leaf node pair is then transformed from its local-coordinate frame to the world-coordinate frame before the more expensive primitive–primitive test is carried out. Using this scheme, the simulation engine is guaranteed to transform only the parts of the objects that are absolutely necessary for the collision check, thereby saving substantially on execution time.

Another interesting observation about this scheme is that, because the simulation engine presented in this book is decoupled from the rendering engine, it does not need to position and orient the objects' primitives in the world-coordinate frame throughout the simulation. After each time step, the simulation engine just needs to position and orient the top-most bounding volume of each object. Of course, it also needs to communicate to the rendering engine the new positions and orientations of the objects that moved since the last simulation time step, so that the rendering engine can itself place and render the objects' primitives at the correct position and orientation. Therefore, as far as the simulation engine is concerned, the cost of moving an object containing several hundred faces, and that does not intersect any other objects in the scene, is the same as moving the top-most bounding volume associated with the object. This in turn reduces even more the simulation engine's execution time.

2.2 Hierarchical Representation of Objects

The hierarchical representations considered in this book are limited to the case when the bounding volumes are either boxes or spheres. Moreover, the object's primitives are assumed to be triangles. This assumption is used not only to speed the primitive–primitive intersection tests, as discussed in Sect. 2.5, but also to simplify building Oriented Bounding Boxes (OBB) trees, as explained in Sect. 2.2.2. The Axis-Aligned Bounding Boxes (AABB) and the Bounding Spheres (BS) representations are not affected by this assumption. These representations are covered in detail in Sects. 2.2.1 and 2.2.3, respectively.

In general, it is not clear which hierarchical representation is best, since collision detection is highly dependent on the relative displacement of the objects being considered. For example, if the objects are close enough to each other, the OBB representation usually function better than the others, in the sense that it considerably reduces the number of primitive–primitive intersection tests owing to its tight fit. On the other hand, if the objects are farther apart, the less expensive bounding volume intersection tests of the AABB and BS representations offer a better choice of hierarchical representation. Modern approaches use hybrid hierarchies with differing representations for its internal nodes based on their proximity to leaf nodes. For example, a hybrid representation can use AABB bounding volumes for its top-most internal nodes providing inexpensive culling of no-colliding regions, and OBB bounding volumes for its leaf and bottom-most internal nodes for improved and more accurate culling.

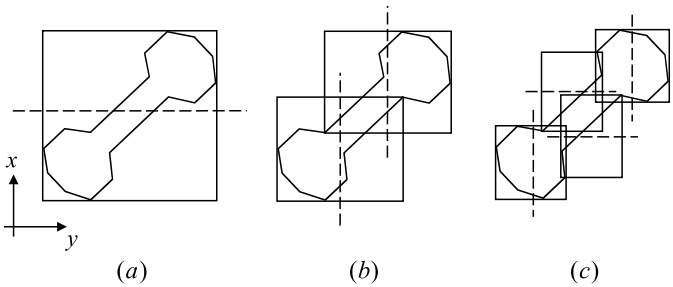

Fig. 2.1 A 2D example of a binary AABB tree. The boxes at each intermediate level are aligned with the axis of the object's local-coordinate frame. The *broken lines* show the partition plane used at each level

Independent of the hierarchical representation used, the tree hierarchy can be constructed in a top-down or bottom-up fashion. In the top-down case, the object's primitives are initially assigned to the top-most bounding volume, which in turn is recursively decomposed into sub-volumes according to some partitioning rule until there is only one primitive or group of primitives assigned to each sub-volume. In the latter case, the sub-division ends with the group of primitives if and only if they can no longer be subdivided according to the partitioning rule. Examples of the top-down approach will be discussed in Sects. 2.2.1 and 2.2.2. In the bottom-up case, the primitives are individually assigned to an initial bounding volume. These bounding volumes are then merged according to some merging rule, until there is only one top-most bounding volume in the tree containing all primitives.

There are several techniques to partition (or merge) bounding volumes into (or from) two or more sub-volumes to form a tree hierarchy. Examples of such partitions are the binary tree (parent has two children), the quad-tree (parent has four children) and the oct-tree (parent has eight children). In this book we limit our analysis to the most common case of building binary tree hierarchies using the top-down approach. Section 2.6 has references to the literature wherein the other possible partitions are covered in detail.

2.2.1 Axis-Aligned Bounding Boxes

In the Axis-Aligned Bounding Box (AABB) representation, the tree hierarchy is constructed from boxes bounding the primitives associated with them, such that the boxes' axes are aligned to the axis of the object's local-coordinate frame. Figure 2.1 illustrates the top-down construction of a binary AABB tree for a simple 2D object.

Initially, the top-most bounding box is constructed by looping through the vertices of all primitives, keeping track of the minimum and maximum values along each axis of the object's local-coordinate frame. The minimum and maximum values will define the lower-left and upper-right corners of the top-most bounding box, respectively.

A partition plane is then selected such that it splits the top-most bounding box into two regions along its longest axis. The intersection point between the partition plane and the longest axis is chosen such that the two regions will be as balanced as possible, that is, with more or less the same number of primitives assigned to each region of the subdivision. The subdivision rule used here is to pass the partition plane through the mean point of all vertices of all primitives associated with the top-most bounding box. The primitives are then assigned to the region in which their midpoint falls.

At each subsequent level, intermediate bounding boxes are constructed from the primitives associated with them, and new partition planes are created to divide the boxes into two regions. The primitives are then assigned to each region and the process recursively continues until there is only one primitive assigned to each region of the subdivision.

In the event all primitives are assigned to just one region (or the subdivision is unbalanced), another partition plane is chosen such that it divides the second longest axis into two regions, passing through the mean point. If this new partition plane still assigns all primitives to just one region, then a last attempt is made with the partition plane dividing the last axis. In the rare case that the group of primitives is assigned to just one region for all three choices of partition plane, the group is said to be indivisible and the current box containing the group becomes a leaf node of the tree. However, in the most common case when the primitives are equally split into both subdivisions, the leaves of the hierarchical tree end up having just one primitive.

2.2.2 Oriented Bounding Boxes

In the Oriented Bounding Box (OBB) representation, the tree hierarchy is constructed from bounding boxes forming a tight fit around the primitives associated with them. In this case, each intermediate bounding box has a different alignment with respect to the object's local-coordinate frame, since their orientation depends on the geometric displacement of their primitives. Figure 2.2 illustrates the top-down construction of a binary OBB tree for the same 2D object considered in the AABB case.

The fact that OBB trees provide a tighter hierarchical representation if compared with AABB trees clearly gives them an advantage over AABB trees when testing for collisions between objects that are close together, since this tightness generally reduces the number of primitive tests to be carried out. However, this comes at the price of having to carry out a more costly overlap test at each intermediate level of the OBB tree, as explained in Sect. 2.5.

The OBB tree construction is much more complex than the simple AABB tree construction, since the orientation of each intermediate bounding box needs to be computed from the set of primitives associated with it. The OBB tree construction algorithm described in this section assumes that the object's primitives are all triangles, that is, that the object's boundary representation is given by triangular faces.

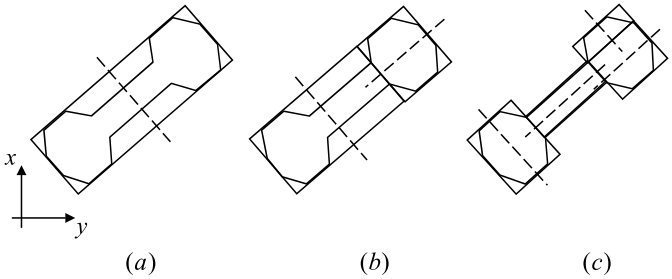

Fig. 2.2 A 2D example of a binary OBB tree. The boxes at each intermediate level provide a tight fit around their primitives. The *broken lines* show the partition plane used at each level

This assumption is especially suited to implementing a simulation engine as described in Chap. 1, since by the time a non-convex object is registered with the simulation engine its convex decomposition is computed (see Appendix F (Chap. 11)) and the faces of each convex polyhedron that make up the object are triangulated. The final internal representation of objects in the simulation engine is therefore made up of triangular faces only.

The main difficulty when computing OBB bounding boxes is the determination of the direction of their axes such that the box provides a tight fit around the vertices of the triangle primitives associated with it. This can be done by considering the mean vector and the covariance matrix of the triangle primitives. The mean vector for each triangle primitive T_k is given by

$$\vec{\mu}_k = \frac{1}{3}(\vec{v}_1 + \vec{v}_2 + \vec{v}_3),$$

where $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$ are the vertices defining the triangle. Each vector $\vec{v}_r$ is described by its components $(v_r)_x$, $(v_r)_y$ and $(v_r)_z$. The mean vector of the vertex set is then

$$\vec{\mu} = \frac{1}{n}\sum_{k=1}^{n} \vec{\mu}_k,$$

with n being the total number of triangles being considered when computing the OBB bounding box.

The elements of the 3×3 covariance matrix of each triangle T_k can be computed as

$$C_{ij} = \frac{1}{3}\big((\bar{p}_1)_i(\bar{p}_1)_j + (\bar{p}_2)_i(\bar{p}_2)_j + (\bar{p}_3)_i(\bar{p}_3)_j\big),$$

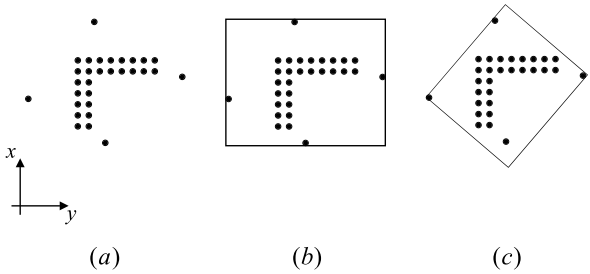

Fig. 2.3 A 2D example of how interior points can degrade the quality of OBB bounding boxes; (**a**) The initial set of points; (**b**) The OBB bounding box created taking all points into account; (**c**) The OBB bounding box created taking into account only the convex hull points

where $i, j \in \{x, y, z\}$ and $\bar{p}_i = (\vec{p}_i - \vec{\mu})$ for $i \in 1, 2, 3$. The covariance matrix of the vertex set is then

$$C_{ij} = \frac{1}{n} \sum_{k=1}^{n} (C_k)_{ij},$$

with $(C_k)_{ij}$ being the $\{ij\}$ element of the covariance matrix associated with the kth triangle.

Since the covariance matrix is a real symmetric matrix, its eigenvectors are guaranteed to be mutually orthogonal. Moreover, two of its three eigenvectors are the axes corresponding to the maximum and minimum variance of the vertices' coordinates. Therefore, if we use the eigenvectors of the covariance matrix as a base, we can determine a tight-fitting bounding box by transforming all vertices to this base and computing the AABB box of the transformed vertices. In other words, the OBB bounding box has the orientation of the eigenvector base and a size that bounds the maximum and minimum coordinates of the transformed vertices.

It is important to notice that the direction of the eigenvectors of the covariance matrix is influenced, not only by the vertices that define the maximum and minimum coordinates, but by *all* vertices being considered. This may cause problems because interior vertices, which should not affect the bounding-box computation, can influence the direction of the eigenvectors. For example, a large number of interior vertices concentrated in a small area can cause the eigenvectors to align with them, instead of aligning with the boundary vertices, thereby creating a low-quality OBB bounding box. This is illustrated in Fig. 2.3.

The computation of the covariance matrix should therefore take into account *only* the boundary vertices of the vertex set. It should also be immune to clusters of boundary vertices, since they will tend to influence the direction of the eigenvectors in the same manner clusters of interior vertices do.

Interior vertices can be avoided if we consider only the vertices that are in the convex hull of the vertex set.[2] Clusters of boundary vertices can be ignored if we compute the mean vector and covariance matrix over the surface of the convex hull,

[2]The computation of the convex hull of a vertex set is described in Sect. 2.2.4.

as opposed to its vertices. This can be done as follows. The area A_k of each triangular face T_k of the convex hull can be computed directly from its vertices, given by

$$A_k = \frac{1}{2}\left|(\vec{v}_2 - \vec{v}_1) \times (\vec{v}_3 - \vec{v}_2)\right|.$$

The total convex hull area A_t is then

$$A_t = \sum_{k=1}^{n_k} A_k,$$

where n_k is the total number of triangular faces in the convex hull.

The mean vector $\vec{\mu}_t$ associated with the convex hull, weighted by the total convex hull area, is obtained from

$$\vec{\mu}_t = \frac{\sum_{k=1}^{n_k} A_k \vec{\mu}_k}{\sum_{k=1}^{n_k} A_k} = \frac{\sum_{k=1}^{n_k} A_k \vec{\mu}_k}{A_t}.$$

The elements $(C_k)_{ij}$ of the 3×3 covariance matrix of each triangular face T_k, also weighted by the total convex hull area, are given by

$$(C_k)_{ij} = \frac{A_k}{12A_t}\big(9(\mu_k)_i(\mu_k)_j + (v_1)_i(v_1)_j + (v_2)_i(v_2)_j + (v_3)_i(v_3)_j\big).$$

Finally, the elements $(C_t)_{ij}$ of the 3×3 covariance matrix associated with the convex hull are computed from the elements $(C_k)_{ij}$ of the covariance matrix of each of its triangular faces as

$$(C_t)_{ij} = \left(\sum_{k=1}^{n_k}(C_k)_{ij}\right) - (\mu_t)_i(\mu_t)_j.$$

Having determined the covariance matrix, we proceed by computing its associated eigenvectors using one of several methods available for computing eigenvalues and eigenvectors of real symmetric matrices. Section 2.6 has pointers to the literature describing such methods. The OBB axis will be aligned with the direction of the eigenvectors and its dimensions will be given by the extremal vertices along each axis.

2.2.3 Bounding Spheres

In the Bounding Sphere (BS) representation, the tree hierarchy is constructed from minimum-radius bounding spheres encapsulating the primitives associated with them. Figure 2.4 illustrates the top-down construction of a binary BS tree for the same 2D object considered in the AABB and OBB cases.

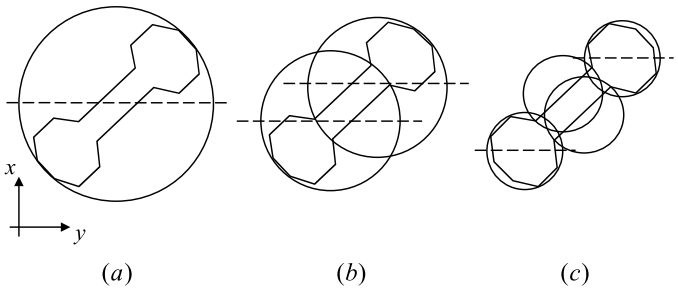

Fig. 2.4 A 2D example of a binary BS tree. The *broken lines* represent the partition plane used at each level

The BS hierarchical tree is usually of poorer quality than its OBB and AABB counterparts, with respect to the tightness of the decomposition. However, its overlap test is undoubtedly the easiest and fastest to carry out (see Sect. 2.5), thus giving this representation an advantage over the others for quick rejection tests.

In this section, we present a method for finding a near-optimal bounding sphere from the set of primitives associated with it. The sphere calculated using this method is usually slightly larger than the minimum-radius sphere, but this inaccuracy is offset by the efficiency of the method.

The bounding sphere computation is carried out in two passes through the list of vertices of all primitives associated with it. The first pass is used to estimate the initial center and radius of the sphere. The second pass goes through each vertex in the list and checks whether it is included in the sphere. If it is not included, then the sphere is enlarged to include it. At the end, the center and radius of the near-optimal bounding sphere are determined.

In the first pass, we loop through the list of all vertices to obtain the following six points.

1. The point with maximum x.
2. The point with minimum x.
3. The point with maximum y.
4. The point with minimum y.
5. The point with maximum z.
6. The point with minimum z.

From these six points, we select the two that are farthest apart. These two points will define the first approximation of the diameter of the bounding sphere. The center of the sphere is assumed to be at their midpoint.

In the second pass, we loop again through the list of all vertices, and for each vertex, we compare the square of its distance to the center with the square of the current radius of the bounding sphere. If the distance is smaller than the radius, then the vertex is inside the sphere and we proceed to the next vertex in the list. Otherwise, we adjust the sphere's radius and center as follows.

Let $\vec{v}_i$ be the current vertex being tested against the bounding sphere, and falling outside it. Let $\vec{c}$ be the center of the bounding sphere, r be its radius and $\vec{p}$ be the point in the sphere diametrically opposed to $\vec{v}_i$ (see Fig. 2.5(a)).

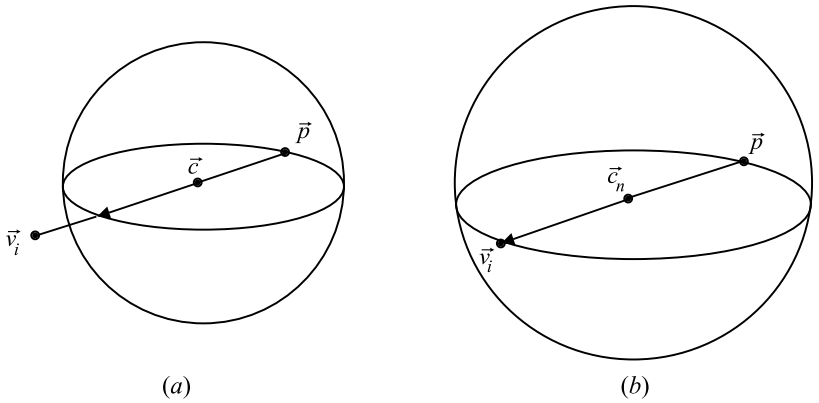

Fig. 2.5 Efficient, incremental computation of a bounding sphere for a given vertex set; (**a**) Vertex $\vec{v}_i$ falls outside the sphere, and therefore the sphere needs to be enlarged to bound it as well; (**b**) The sphere is augmented such that $\vec{v}_i$ and $\vec{p}$ define its new diameter

Let d be the distance between $\vec{v}_i$ and $\vec{c}$, that is

$$d = \sqrt{\left((v_i)_x - c_x\right)^2 + \left((v_i)_y - c_y\right)^2 + \left((v_i)_z - c_z\right)^2}.$$

The enlarged sphere is then computed from the current sphere such that $\vec{v}_i$ and $\vec{p}$ become the new diameter, as shown in Fig. 2.5(b). The new center $\vec{c}_n$ and radius r_n of the enlarged sphere are given by

$$r_n = \frac{r + d}{2}$$
$$\vec{c}_n = \frac{r\vec{c} + (d - r)\vec{v}_i}{d}.$$

This process continues until all vertices are checked for inclusion against the bounding sphere.

Having determined the top-most bounding sphere, a partition plane is chosen such that it passes through the median point of all vertices of all primitives associated with the bounding sphere. The partition plane subdivides the bounding sphere into two regions, and the primitives are assigned to each region following the same rules used on the AABB and OBB cases, namely, the primitive is associated with the region that contains its midpoint. The subdivision continues until there is only one primitive assigned to each bounding sphere, or the primitives cannot be split, in which case the group of primitives is assigned to the bounding sphere.

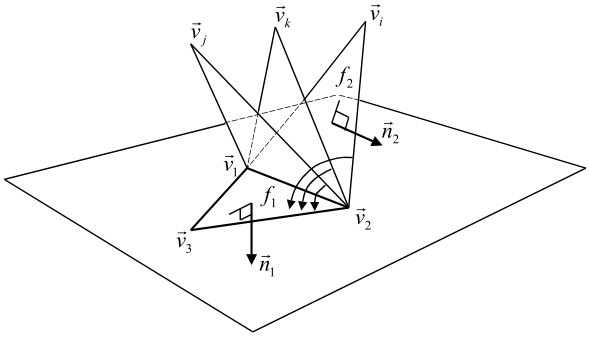

Fig. 2.6 Determining the neighbor face that shares edge e_1 with f_1. The selected vertex $\vec{v}_i$ will define a face f_2 that forms the largest convex dihedral angle with f_1. The ordering of the vertices defining the new face should be chosen so that the normal vector of the new face always points towards the outside of the object. Because we are using the right-hand coordinate system, the correct order is $(\vec{v}_1, \vec{v}_i, \vec{v}_2)$

2.2.4 Convex Hull

The convex hull can be used not only to provide a hierarchical representation of the object's primitives as a tree of convex polyhedra, but also as an intermediate step for computing other types of representations, such as the OBB trees already covered in Sect. 2.2.2.

The convex hull of a given vertex set S is defined as being the smallest convex set containing S. There are several algorithms and methods that can be used to compute the convex hull in 2D, 3D, or even higher dimensions. In this section, we focus on the *gift wrapping* method, which is intuitive, easy to visualize in 3D, simple to implement, and applicable to higher-dimensional spaces.

The basic idea of the gift-wrapping method consists of imagining folding a piece of paper around the primitives being considered. We start with a face that is guaranteed to be in the convex hull, and loop through its edges determining its neighbor faces that are also part of the convex hull. The algorithm then proceeds looping through the edges of the new neighbor faces until all faces are discovered and the convex hull is completely determined. All faces will be discovered whenever the list of edges to be checked is empty.

Given a set of vertices $S = \{\vec{v}_1, \ldots, \vec{v}_n\}$, let's assume that the triangular face f_1 defined by vertices $(\vec{v}_1, \vec{v}_2, \vec{v}_3)$ is the starting face guaranteed to be in the convex hull. According to the high-level description of the algorithm presented in the previous paragraph, we need to loop through the edges of face f_1 and determine its associated neighbor faces that are also in the convex hull. A face is said to be in the convex hull if all vertices of S that are not vertices of the face lie on the same side of the plane defined by the face. Since we are using the right-hand coordinate system in our simulation engine, we want all vertices of S to lie on the inside region of the plane. More specifically, we want to construct each face of the convex hull such that its normal is always pointing outwards, as illustrated in Fig. 2.6.

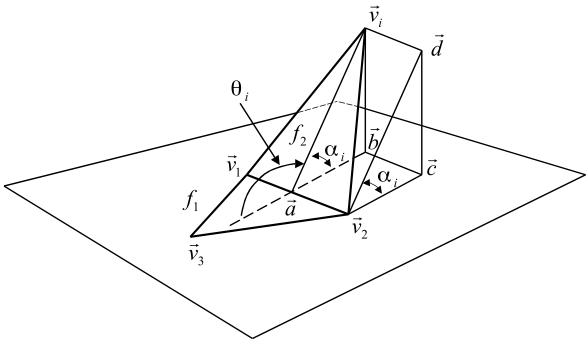

Fig. 2.7 The internal dihedral angle θ_i associated with vertex $\vec{v}_i$ at edge e_1 defined by vertices $(\vec{v}_1, \vec{v}_2)$, shown as the exterior angle at vertex $\vec{a}$ of triangle $(\vec{a}, \vec{b}, \vec{v}_i)$. Notice that vertex $\vec{b}$ is the projection of $\vec{v}_i$ on the plane of face f_1

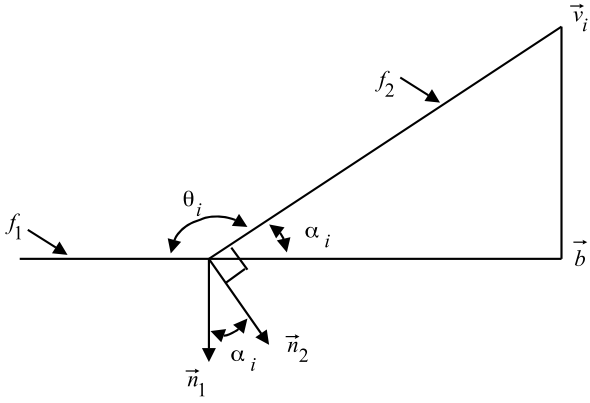

Fig. 2.8 The interior angle α_i at vertex $\vec{a}$ can be obtained from the dot product of the face normals $\vec{n}_1$ and $\vec{n}_2$ associated with faces f_1 and f_2, respectively

Let's consider, for example, the determination of the neighbor face f_2 that shares edge $e_1 = (\vec{v}_1, \vec{v}_2)$ with f_1. We want to find the vertex $\vec{v}_i \in S$, with $i \neq 1, 2$, such that the triangular face f_2 defined by $(\vec{v}_1, \vec{v}_i, \vec{v}_2)$ forms the largest convex internal dihedral angle at edge e_1. Figure 2.7 shows how the internal dihedral angle can be computed.

Let θ_i be the internal dihedral angle associated with vertex $\vec{v}_i$ at edge e_1. Let $\vec{n}_1$ and $\vec{n}_2$ be the normal vectors of faces f_1 and f_2, respectively. By construction, since the normals at each face are pointing outwards, their dot product gives the cosine of $(\pi - \theta_i)$ (see Fig. 2.8). The dihedral angle can then be computed directly from

$$\theta_i = \pi - \arccos(\vec{n}_1 \cdot \vec{n}_2).$$

We select the vertex $\vec{v}_i$ corresponding to the maximum θ_i, and add face f_2 to the list of convex hull faces. The ordering of the vertices defining the new face should be chosen so that the normal vector of the new face always points towards the outside of the object. Because we are using the right-hand coordinate system, the correct order is $(\vec{v}_1, \vec{v}_i, \vec{v}_2)$. The edges of f_2 are then added to the list of edges that need to be checked, so that the algorithm can compute the convex hull faces that share these edges with f_2. It is important to notice that the algorithm assumes each edge

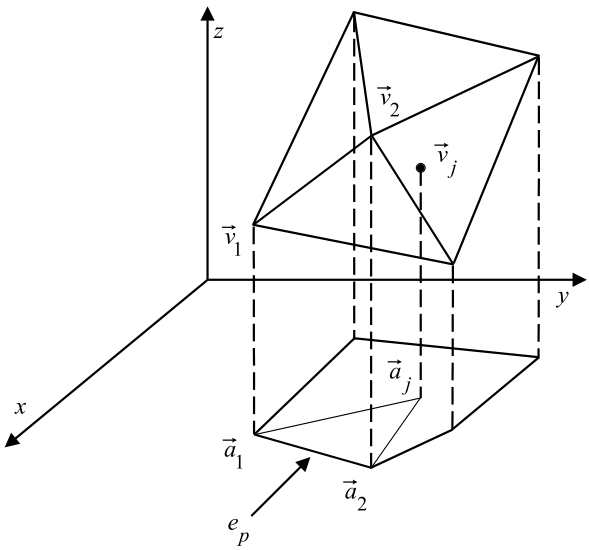

Fig. 2.9 The first edge of the starting face is computed using the projection of the vertices on the xy-plane. Here the problem is reduced to its 2D counterpart

is shared by exactly two faces. Therefore, every time a new edge is added to the list of edges that need to be checked, we should first check whether the edge is already in the list. If the edge is already in the list, then one of the faces that contains this edge was already discovered in some previous step, and the other face that contains this edge has just being discovered. In this case, there is no need to check for this edge because both faces that share the edge are already included in the convex hull. Therefore, the edge can be removed from the list. Otherwise, the edge should be added to the list.

Up till now we have assumed the existence of a starting face that is guaranteed to be in the convex hull, and have determined all other faces from it. The only step we still have to describe is how the first face of the convex hull is computed. The computation of the vertices of the first face is incremental, in the sense that we compute one of them at a time. We start with one vertex that is guaranteed to be in the convex hull, then use it to determine the second vertex, thus forming an edge of the starting face. We then use the edge to determine the third vertex that makes up the first face. Having the first face, we proceed as explained before and determine all other convex hull faces.

The first face of the convex hull is computed as follows. Consider the projection of all points on the xy-plane, as shown in Fig. 2.9. Let $\vec{a}_1$ be the vertex with the lowest projected y-coordinate value. This vertex is guaranteed to be in the convex hull, since all other points of the vertex set will lie on the same half-space defined by a plane orthogonal to the y-axis (i.e., parallel to the xz-plane), passing through vertex $\vec{v}_1$. Therefore, vertex $\vec{v}_1$ is one of the vertices of the starting face.

The second vertex of the starting face can be found by looping through the projected vertices and selecting a vertex $\vec{a}_2$ such that all other projected vertices lie to the left of the edge $e_p = (\vec{a}_1, \vec{a}_2)$. We can determine whether the projected vertex $\vec{a}_j$

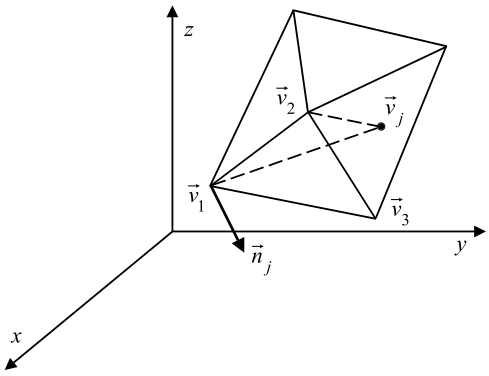

Fig. 2.10 The starting face is obtained by connecting a third vertex to the starting edge, so that all other vertices lie on the negative half-space defined by the plane that contains the face

lies to the left or right of edge e_p by considering the sign of the area of the triangle defined by $(\vec{a}_1, \vec{a}_2, \vec{a}_j)$. If the area is positive, then the vertices are in counterclockwise order and the projected vertex $\vec{a}_j$ lies to the left of edge e_1. Otherwise, the projected vertex $\vec{a}_j$ lies to the right of edge e_p.

The area A of the projected triangle $(\vec{a}_1, \vec{a}_2, \vec{a}_j)$ can be quickly computed from the vertices' coordinates

$$A = \frac{1}{2} \begin{vmatrix} (\vec{a}_1)_x & (\vec{a}_2)_x & (\vec{a}_j)_x \\ (\vec{a}_1)_y & (\vec{a}_2)_y & (\vec{a}_j)_y \\ 1 & 1 & 1 \end{vmatrix}.$$

Finally, the third vertex of the starting face can be obtained by considering the triangular faces f_j defined by vertices $(\vec{v}_1, \vec{v}_j, \vec{v}_2)$ in 3D space. The order of the vertices defining the triangular face f_j is such that the normal points to the outside of the convex hull.[3] Here, the third vertex $\vec{v}_3$ is selected such that all other vertices lie in the negative half-space defined by the plane that contains the triangular face $(\vec{v}_1, \vec{v}_3, \vec{v}_2)$ (see Fig. 2.10).

Let $\vec{n}_j$ be the normal of the plane defined by the triangular face $(\vec{v}_1, \vec{v}_j, \vec{v}_2)$, and let d_j be the plane constant computed as

$$d_j = \vec{n}_j \cdot \vec{v}_1.$$

Vertex $\vec{v}_p$ will lie on the negative half-space of the plane provided

$$(\vec{n}_j \cdot \vec{v}_p) < d_j.$$

Having the first face, we proceed as explained and compute all other convex hull faces of the polyhedron.

[3] Recall that we are using the right-hand coordinate system.

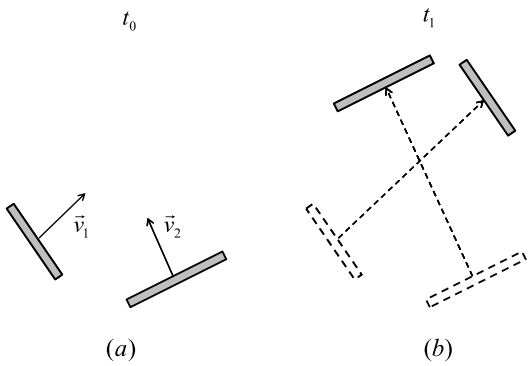

Fig. 2.11 (**a**) Two fast moving objects at t_0; (**b**) Even though they do not collide at t_1, they do pass through each other during their motion from t_0 to t_1

2.3 Hierarchical Representation for Continuous Collision Detection

In general, the collision detection between objects moving for a time interval $[t_0, t_1]$ is performed by a pairwise intersection test of the objects at t_1. Depending on the objects' shape and velocity, it is possible that some pairs of objects end up not intersecting at t_1 even though their paths crossed each other during their motion. This is usually the case for thin or fast moving objects, as illustrated in Fig. 2.11.

The continuous collision detection is suitable to handle these special cases because it takes into account the motion of the objects from t_0 to t_1, not just their position and orientation at t_1. It still uses the hierarchical representations of the objects to speed computations, but they need to be modified to bound the entire motion from t_0 to t_1 in *world-coordinate frame*. Notice that the standard procedure is to construct the hierarchies once in their local-coordinate frame at the time the objects are registered with the simulation engine, and incrementally transform their nodes from local- to world-coordinate frame as they are checked for intersections in world space. In the continuous collision case, the hierarchies need to be constructed in the world-coordinate frame to take into account the objects' motion in world space. Theoretically, we need to re-build the hierarchies at the end of each time interval using the top-down approach already explained in Sect. 2.1, but replacing each primitive with a bounding volume that contains its world-coordinate positions at both t_0 and t_1. However, the significant performance hit it takes on the simulation to rebuild the hierarchical trees at every time interval deems this approach impractical, especially when the number of objects being simulated in large (i.e., in the thousands).

A more efficient approach adopted in this book is to *refit* the hierarchical trees originally built in their local-coordinate frames, but with the motion information in world space. The efficiency comes from the fact that the process of refitting a hierarchical tree retains the parent–children relationship obtained when the tree was initially built, so all intermediate computations to determine the bounding volumes of internal nodes in a top-down fashion are forgone. When refitting a tree, the leaf nodes are updated first to bound their primitive's motion from t_0 to t_1, and then their parent (internal) nodes are updated to bound the volume of their children. It

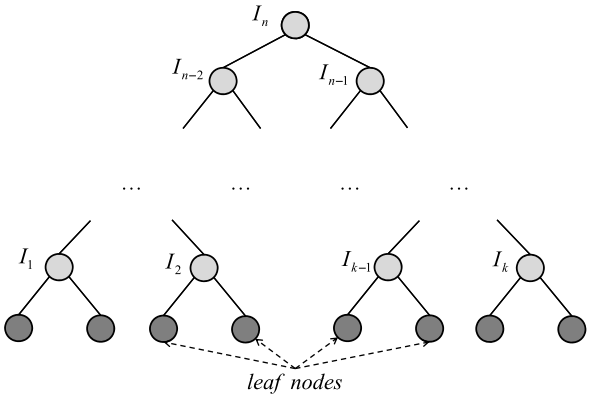

Fig. 2.12 Hierarchical tree representation of an object

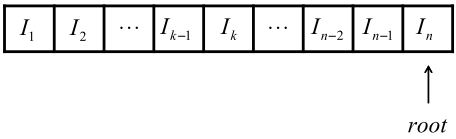

Fig. 2.13 Array representation of the internal nodes in Fig. 2.12, sorted by decreasing depth (i.e., distance to the root node). The root node is positioned at the last entry because it is the only node that does not have a parent

is important that an internal node is updated only after its children are updated. This requires special attention when building the hierarchical tree for the first time, to make sure the internal nodes are stored in decreasing order with respect to their depth (i.e., their distance to the root node). Figure 2.13 shows an array representation of all internal nodes of the hierarchical tree shown in Fig. 2.12, sorted by decreasing depths. A node located at the k-th entry is guaranteed to have its parent node located at an entry j with $j > k$, that is, by updating the internal nodes using their order in the sorted array, we can guarantee the children nodes will always be updated before their parents.

As far as performance is concerned, refitting a tree is about an order of magnitude faster than rebuilding it. A more detailed discussion on the advantages and disadvantages of refitting a hierarchical tree can be found in the references provided in Sect. 2.6.

2.4 Hierarchical Representation of the Simulated World

Even though the use of hierarchical representations does speed the collision-detection phase, they themselves do not provide mechanisms to take advantage of the time coherence between consecutive frames in a simulation. For instance, the fact that two or more objects are farther apart, such that their top-most bounding

volumes do not intersect, should be exploited in the following simulation time steps to avoid unnecessary collision checks between these objects. The hierarchical representation does minimize the time spent on such unnecessary collision checks, since they are usually dismissed after the top-most bounding volumes are checked against each other. However, the time expended on these unnecessary checks can be significant, especially when the simulation contains several thousand objects.

The idea is then to create a partition of the simulated world into cells, and assign objects to the cells in which their top-most bounding volume intersects. Objects that are assigned to the same cell can be potentially colliding and therefore should be checked for geometric intersections between their hierarchical representations. On the other hand, objects that have no cells in common are clearly distant from each other and should not be checked for collisions at all.

The simulated world considered in this book is assumed to be bounded by a box that defines the maximum and minimum spans along each coordinate axis. The cell decomposition is then a partition of the box into sub-volumes that may or may not contain objects during the simulation.

There are two important issues that should be taken into account when decomposing the simulated world into cells. First, the cell decomposition should be simple, that is, should have simple geometry such that the cost of updating the cells that intersect each moved object is negligible compared with the cost of checking for collisions between their hierarchical representations. This issue is addressed in Sect. 2.4.1, where the simulated world is subdivided into boxes of uniform size. Second, the size of each cell directly affects the efficacy of the decomposition. For instance, a too-small size will assign several cells to each object, making it more expensive to update the list of occupied cells after each simulation time step. On the other hand, a too-large size will assign several objects to the same cell and a large number of unnecessary collision checks between their hierarchical representations may be carried out. This issue is addressed in Sect. 2.4.2, where the uniform-grid approach presented in Sect. 2.4.1 is extended to a multi-level grid that better fits the different sizes of objects being simulated.

2.4.1 Uniform Grid

Uniform-grid decomposition, as its name implies, subdivides the bounding box of the simulated world into cubic cells of same size along each axis of the world-coordinate frame. A simple uniform-grid decomposition of an hypothetical world is shown in Fig. 2.14.

In the uniform-grid decomposition, the dimension of the cubic cells plays an important role in minimizing the number of unnecessary objects to cell assignments, and in maximizing the overall efficacy of the simulation. Intuitively, the size of each cell should be:

- Large enough to allow objects to rotate and translate for a while without leaving the cell, thus minimizing the number of dynamic updates of objects assigned to each cell;

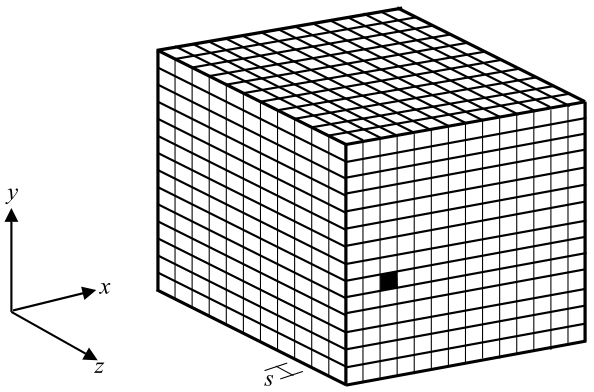

Fig. 2.14 A simple uniform-grid decomposition of an hypothetical simulated world with a resolution of fourteen boxes along each dimension. Each cell is identified by the index on its lower-left corner vertex. For example, the first cell is indexed as cell $(0, 0, 0)$. The shaded cell is indexed as cell $(2, 5, 13)$

- Small enough to have as many objects as possible assigned to different cells, thus minimizing the pair-wise collision checks between the object's hierarchical representations.

Let d_i be the maximum diameter of the top-most bounding volume of object i. For example, if the bounding volume is a box, then the maximum diameter is the distance between the two diagonally opposing vertices defining box. If the bounding volume is a sphere, then the maximum diameter is equal to the diameter of the sphere. The average maximum diameter of the objects being simulated is then

$$\bar{d} = \frac{1}{n} \sum_{i=0}^{n} d_i,$$

where n is the total number of objects.[4] The size b of each cell in the uniform-grid decomposition is also given by the cell's maximum diameter, and can be related to the average maximum diameter of the objects being simulated as

$$\frac{\bar{d}}{b} = k, \tag{2.1}$$

with $k \geq 1$. The variable k is used to adjust the size of the cell with respect to the average maximum diameter of the objects being simulated. As a rule, we suggest using $k = 2$, that is, the size of each cell in the uniform grid is twice the average maximum diameter. The rationale behind this choice is as follows. If all objects had the average size, then we could have up to eight objects in each cell (two objects touching each other along each dimension), giving some room to an object to move within the same cell. Also, objects that are farther apart by more than twice their average size are guaranteed not to be in the same cell. On average, this choice gives us a reasonable trade-off between the number of objects assigned to each cell and

[4]The size of particles in a particle system is not taken into account during this computation because particles are usually considered as point mass, as explained in detail in Chap. 3.

the number of pair-wise collision checks carried out at each time step if the objects' sizes are close to the average. However, if the objects' sizes vary by orders of magnitude, then a more sophisticated approach, such as the one presented in Sect. 2.4.2, should be used.

Having selected the size of the boxes in the decomposition, the next step is to provide an efficient mechanism to keep track only of cells that have at least one object assigned to them, as opposed to allocating memory to all cells. Clearly, the latter approach is not advisable for cases when the size of each cell is orders of magnitude smaller than the size of the simulated world, since the number of cells along each axis would be huge, and the memory needed for a subdivision containing n cells is n^3. The idea here is to use a hash table to keep track of the occupied cells. There are many ways this hash table of cells can be constructed, and some may be more effective than others, depending on the specifics of the simulation being considered. However, as a rule, we suggest using a hash table of size n, where the key is the sum of the indexes of the cell along each axis. This will have the effect of assigning one slice (plane) of the grid decomposition to each hash-table entry.

The update mechanism using the hash table is used to efficiently detect pairs of potentially colliding objects. Initially, the top-most bounding volume of each object is checked against the cell decomposition. Each cell that intersects the object's bounding volume is added to the hash table of cells. If the cell was already added to the hash table, then there are at least two objects assigned to this cell, and a pointer to this cell is added to a list of cells that need to be checked for collisions. At the end, after all objects are checked against the cell decomposition, another list of potentially colliding objects is created from the list of cells that needs to be checked for collisions. The former list contains pairs of objects that occupy the same cell. For each of these pairs, the more expensive collision check using their hierarchical representation is carried out.

The cells of the uniform-grid decomposition that intersect an object's bounding volume can be efficiently determined if we consider an AABB bounding box of the object's bounding volume in the world-coordinate frame. Notice that this selection is independent of the hierarchical tree representation of the objects. The AABB bounding box is aligned with the world coordinate frame, as are all cubic cells in the uniform-grid decomposition. The box–box intersection test between boxes that have their axes aligned is extremely fast and can be used to determine the actual cubic cells in the decomposition that need to be checked for intersection with the object's bounding volume. Figure 2.15 illustrates this situation for an object using the bounding-sphere representation. The AABB bounding box of the bounding sphere is used to efficiently locate the cells in the decomposition that need to be checked for intersection with the object's bounding sphere, as opposed to checking the intersection of every cell with the object's bounding sphere.

As objects translate and rotate during the simulation, their top-most bounding volume will move. This movement may cause the bounding volume to no longer intersect some of the cells the object is assigned to, and may also intersect new cells that did not have the object on their list. Therefore, the list of objects assigned to each cell needs to be updated after each simulation time step to reflect these changes.

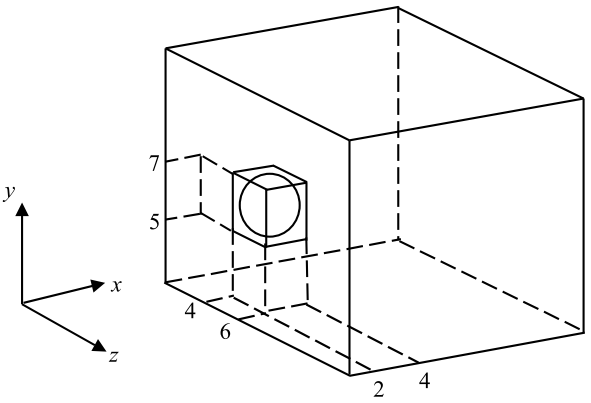

Fig. 2.15 The AABB bounding box of the object's bounding sphere is used to quickly determine which cells of the decomposition need to be checked for intersection with the bounding sphere. In this case, the bounding sphere will be checked further for intersections with cells (2, 5, 4), (2, 6, 4), (3, 5, 4), (3, 6, 4), (2, 5, 5), (2, 6, 5), (3, 5, 5) and (3, 6, 5)

This update can be efficiently implemented using coherence between simulation time steps, as follows.

Throughout the simulation, each object keeps track of the indexes of the cells that intersect its bounding volume. At each time step, a new list of indexes of cells that intersect the object's bounding volume is generated. This new list is then compared with the old list. If the new list is the same as the old list, then the object's cell assignment remains the same as in the previous time step, and nothing else needs to be done. If there are cells on the new list that are not on the old list, then we search for these cells in the hash table of cells. If we find the cell in the hash table, then we add a reference to this object and raise the cell's internal counter. Otherwise, we create an entry for the cell in the hash table and set its internal counter to one. Finally, if there are boxes on the old list that are no longer on the new list, then the reference to this object should be removed from the cell's entry in the hash table, and its internal counter is subtracted by one.

The cell's internal counter is used to keep track of the number of objects currently intersecting the cell. The first time the counter is set to two, a reference to this cell is added to the list of cells that contain potentially colliding objects. The counter can be set to values greater than two, but as long as it has at least two, the reference to this cell will be kept in the list. The reference to this cell is removed from the list when the counter first is reduced from two to one. The cell is removed from the hash table of cells when the counter is set to zero.

2.4.2 Multi-level Grid

If the size of the objects being simulated differs by orders of magnitude, the efficiency of the uniform-grid approach can be improved by extending it to a multi-level grid. The idea is to group at the same level, objects with sizes of the same order of magnitude such that each level can be treated as an uniform grid in itself. The advantage is that each level attempts to maximize the efficiency of its uniform grid, since it is guaranteed to have objects of similar size. There are three important issues that need to be addressed when using this method.

- How many levels should be chosen for a given set of objects?
- What should be the size of the cells at each level?
- How the levels are related so that collisions between objects assigned to different levels can be efficiently detected?

In the uniform-grid case, since we have a single level, the size of the cell is determined from Eq. (2.1) as a multiple of the average size of the maximum diameter of the objects in the simulation. In the multiple-level case, object i is assigned to level j if

$$k_{min} \leq \frac{d_i}{L_j} \leq k_{max}, \tag{2.2}$$

where d_i is the maximum diameter of the top-most bounding volume of object i, k_{min} and k_{max} are user-definable constants, and L_j is the size of the cells at level j, such that

$$0 < L_1 < L_2 < \cdots < L_j < \cdots < L_m$$

for $1 \leq j \leq m$. The idea is to assign object i to the largest level j such that Eq. (2.2) is satisfied. In other words, the largest objects are assigned to the largest boxes (largest levels), so that objects at level $(j+1)$ have diameters greater than objects at level j. The constants k_{min} and k_{max} are used to relate the size of the cells at different levels. They must satisfy

$$0 < k_{min} < 1$$
$$k_{max} \geq 1.$$

Let d_{min} and d_{max} be the minimum and maximum diameters of all objects in the simulation. Clearly, the objects associated with d_{min} (the smallest objects) should be assigned to level 1 (the lowest level). This is done by substituting d_{min} and L_1 into Eq. (2.2), that is

$$k_{min} \leq \frac{d_{min}}{L_1} \leq k_{max}.$$

If we make

$$k_{min} = \frac{d_{min}}{L_1},$$

then we have that the size of the cells at the lowest level is given by

$$L_1 = \frac{d_{min}}{k_{min}}. \tag{2.3}$$

The objects associated with d_{max} should be assigned to the largest level m. This is done by substituting d_{max} and L_m into Eq. (2.2), and making

$$\frac{d_{max}}{L_m} = k_{max},$$

that is

$$L_m = \frac{d_{max}}{k_{max}}. \tag{2.4}$$

Because we want the level assignment to be continuous, we need to make sure the maximum value at level j is equal to the minimum value at level $(j+1)$, that is

$$k_{max} L_j = k_{min} L_{j+1}. \tag{2.5}$$

Equation (2.5) relates the size of the cells at two consecutive levels. We can use this equation to recursively compute the size of the cells at level j as a function of the size of the cells at level 1, as follows:

$$\begin{aligned} L_2 &= \frac{k_{max}}{k_{min}} L_1 \\ L_3 = \frac{k_{max}}{k_{min}} L_2 &= \left(\frac{k_{max}}{k_{min}}\right)^2 L_1 \\ &\cdots \\ L_j &= \left(\frac{k_{max}}{k_{min}}\right)^{j-1} L_1. \end{aligned} \tag{2.6}$$

Because we have the size of the boxes at the first level L_1 and the largest level L_m given by Eqs. (2.4) and (2.5), respectively, we can substitute these equations into (2.6) and compute the number of levels m needed as a function of k_{min}, k_{max}, d_{min} and d_{max}. We have

$$\frac{d_{max}}{k_{max}} = \left(\frac{k_{max}}{k_{min}}\right)^{m-1} \frac{d_{min}}{k_{min}},$$

which gives

$$m = \left\lceil \log_{(\frac{k_{max}}{k_{min}})} \left(\frac{d_{max}}{d_{min}}\right) \right\rceil. \tag{2.7}$$

Figure 2.16 shows an example of a multi-level grid assignment. At each level j, the simulated world is subdivided in a uniform grid with boxes of size L_j.

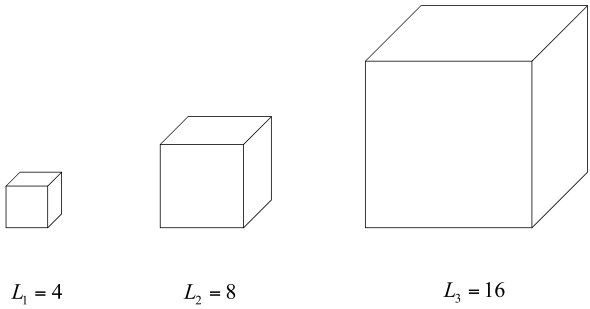

Fig. 2.16 An example of a multi-level grid assignment for $k_{max} = 1$, $k_{min} = 0.5$, $d_{max} = 16$ and $d_{min} = 2$. The maximum number of levels to be used and their sizes can be directly computed from Eqs. (2.7) and (2.6), respectively. In this case, $m = 3$, $L_1 = 4$, $L_2 = 8$ and $L_3 = 16$

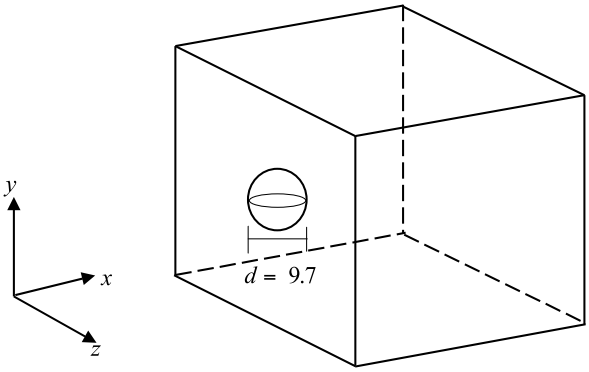

Fig. 2.17 An example of the multi-level grid assignment of an object to level 2, in the simulated world of Fig. 2.16, containing 3 levels

Therefore, the multi-level grid has one hash table of cells for each level, and the sizes of their cells are given by Eq. (2.6). The update mechanism for each hash table is the same as that used in the uniform grid, since we do have a uniform grid at each level. Consider, for example, the bounding sphere of an object with center at (5, 5, 5) and maximum diameter $d = 9.7$ (see Fig. 2.17). For the multi-level grid of the simulated world shown in Fig. 2.16, according to Eq. (2.2), the object should be assigned to level 2. Within level 2, the cells that intersect the object's bounding volume are computed the same way as shown in Fig. 2.15 for the uniform-grid case.

The only remaining issue is how this scheme can be used to efficiently detect potential collisions between objects assigned to different levels of the grid decomposition. We address this issue by adding a reference to the object, not only to the cells that intersect the object at its level, but also to all other cells that intersect the object at levels greater than its level. For example, an object assigned to level j will have its reference added to all cells that intersect its bounding volume at levels j, $(j + 1)$, ..., m. In the case shown in Fig. 2.17, the cells at levels 2 and 3 that intersect the object's bounding volume will keep a reference to this object (see Figs. 2.18 and 2.19). Using this scheme, two objects b_1 and b_2 assigned to levels L_{b_1} and L_{b_2} can potentially collide if and only if there exist at least one cell at level $\max(L_{b_1}, L_{b_2})$ that has a reference to both of them. This situation is illustrated in the one-dimensional case shown in Fig. 2.20.

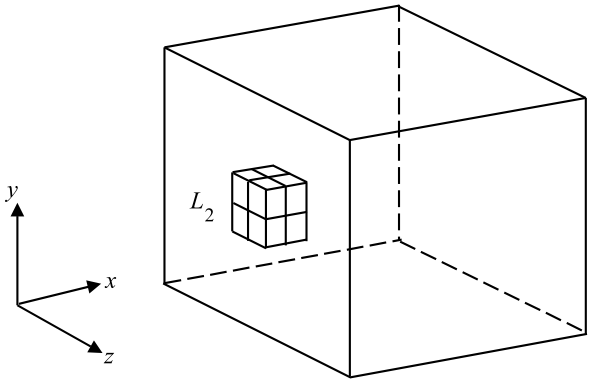

Fig. 2.18 The cells at level 2 that intersect the object's bounding sphere

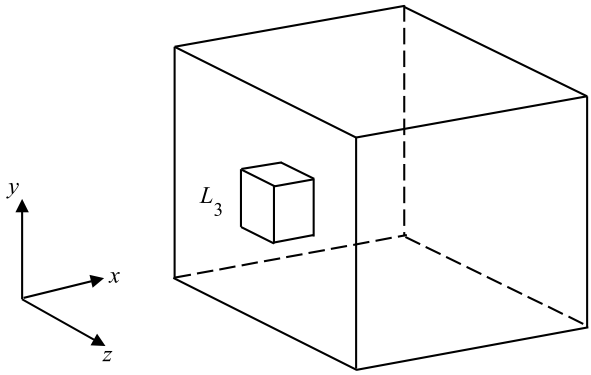

Fig. 2.19 The multi-level grid assignment makes it necessary to determine the cells at level 3 that intersect the object. This is in order to detect potential collisions between this object and other objects assigned only to level 3

In this example, objects b_1 and b_2 are assigned to levels L_1 and L_3, respectively. Since there is a box at level $m = \max(L_1, L_3) = L_3$ that contains a reference to both of them, the objects are added to the list of potentially colliding objects. On the other hand, objects b_3 and b_4 are assigned to levels L_1 and L_2, respectively. Since there are no boxes at level $m = \max(L_1, L_2) = L_2$ that contain a reference to both of them, they are not considered to be on the list of potentially colliding objects, even though there is a cell at level L_3 referring to both of them.

As a rule, we suggest choosing k_{min} and k_{max} such that

$$\frac{k_{max}}{k_{min}} = 2.$$

This choice means the size of the cells will be a power of two times the minimum diameter d_{min}. With this choice, given an object assigned to level j, it is straightforward to ascertain which cells intersect the object at levels $(j + 1), \ldots, m$.

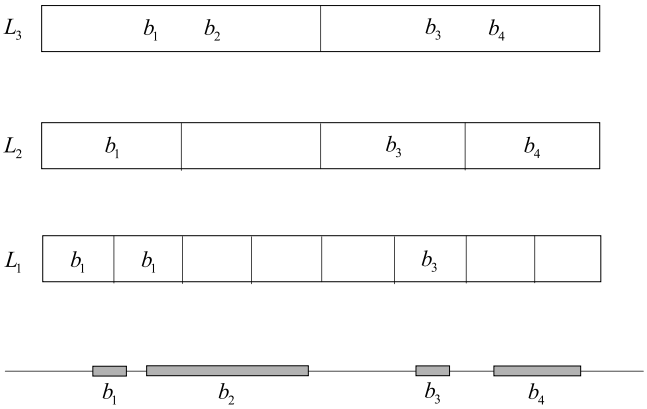

Fig. 2.20 A one-dimensional example of how potential collisions are detected between objects assigned to different levels

2.4.3 Bounding Volume for Continuous Collision Detection

The grid-based representation of the simulated world becomes less efficient for continuous collision detection, where the entire motion of the objects is considered. This is because objects may pass through several grid cells during their motion thus creating a lot of redundant entries in the list of potential collisions obtained from the pairs of objects that share a cell. A better way of representing the simulated world for continuous collision detection is to use a hierarchical tree representation with leaf nodes bounding the objects' entire motion. This is very similar to the hierarchical representation of objects discussed in Sect. 2.3, where leaf nodes contain the motion of their primitives for the entire time interval.

Figure 2.21 shows an example of a system with 3 objects. Their motion for the current time interval is bounded by axis-aligned boxes that are used as leaf nodes in the simulated-world tree representation of this system. The list of potential collision pairs is obtained by self-intersecting the tree, that is, by detecting all pairs of objects that cross each other's paths during their motion.

2.5 Collision Detection Between Hierarchical Representations

Up till now, we have described several types of hierarchical representations that can be used to speed collision detection between objects in a simulation engine, as well as how we can build them using simple primitives. In this section, we shall present efficient algorithms to quickly determine whether two hierarchies or two of their primitives are intersecting.

The primitives of the representations covered in this book are boxes and spheres for the tree hierarchies, and triangles for the faces of the objects. Therefore, we need algorithms for checking the intersection of each possible pair-wise combination of

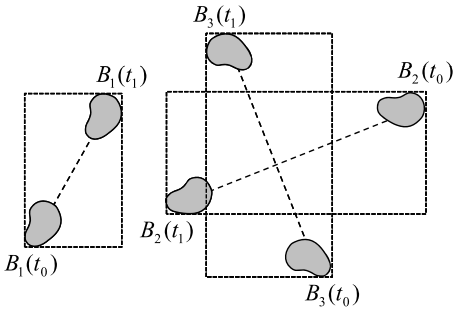

Fig. 2.21 A simple system with 3 objects being simulated. Their motion is bounded by axis-aligned boxes, which are used as leaf nodes for the hierarchical tree representation of the simulated world. Collision candidates are obtained by self-intersecting the world tree. In this example, the pair of objects (B_2, B_3) is a collision candidate for this time interval

such primitives. Moreover, for the triangle–triangle intersection test, we need to go one step further and save pointers to the intersecting triangles. This information will be used by the collision-response module to estimate the collision (or contact) point between the objects, and determine the collision impulses (or contact forces) needed to prevent their interpenetration.

2.5.1 Computing Hierarchy–Hierarchy Intersections

Let H_1 and H_2 be two hierarchical representations of objects that are currently being checked for collisions. Here, the hierarchies are used to help speed the detection of all primitive–primitive pairs that are potentially colliding, without incurring the $O(n^2)$ cost of testing every primitive of one object against all primitives of the other object.

The hierarchy–hierarchy intersection test is carried out in a top-down manner doing pair-wise intersection tests between their internal nodes as needed. A priority queue Q is used to keep track of all internal node pairs that still need to be tested for intersection. This queue is initialized with the pair containing the root nodes of each hierarchy. At each iteration of the intersection test, we remove the front element of Q and check if its corresponding pair of nodes are intersecting. If no intersections are found, then we proceed to the next iteration, that is, the next element in the queue. Otherwise, the pair-wise combination of the intersecting nodes' children are added to the back of Q, so that they too can be checked for intersections. In the special case in which the intersecting nodes have no children, that is, they are leaf nodes of their hierarchies, then their corresponding primitives are added to a list of potentially colliding candidates. The following summarizes this recursive process.

1. Get (and remove) the first element in Q. Let $n_1 \in H_1$ and $n_2 \in H_2$ be the nodes corresponding to this element.

2. Check if the nodes intersect. If they don't intersect, discard this pair and move on to the next element in the queue.
3. The nodes do intersect at this point. Now, check if both nodes are leaves of their trees. If both of them are leaves, then add their primitives to the list of collision candidates. Discard this pair and move on to the next element in the queue.
4. Check if at least one of the nodes is a leaf. If one of the nodes is in fact a leaf, say n_1, then do the following:
   (a) Add the pair $(n_1, (n_2)_{left})$ to the priority queue, where $(n_2)_{left}$ is the child to the left of the node n_2.
   (b) Add the pair $(n_1, (n_2)_{right})$ to the priority queue, where $(n_2)_{right}$ is the child to the right of the node n_2.
5. At this point, both nodes n_1 and n_2 are internal nodes of their hierarchies. The following pairs are added to the priority queue for further intersection tests:
   - $((n_1)_{left}, (n_2)_{left})$,
   - $((n_1)_{left}, (n_2)_{right})$,
   - $((n_1)_{right}, (n_2)_{left})$,
   - $((n_1)_{right}, (n_2)_{right})$.

At the end of this process, we have a list of collision candidates containing pairs of primitives that need to be tested for intersections. The usually more expensive primitive–primitive intersection tests are carried out for all elements in this list. The hierarchies do not intersect if the list of collision candidates is empty, or no pairs of collision candidate primitives end up intersecting.

The fact that the hierarchies are not intersecting at the end of the current time interval does not guarantee that the objects are not colliding. Depending on the dynamic state of the objects and the size of the time interval used, it is possible that one object has moved completely inside the other object. In order to detect such cases, we need to perform an additional point-in-object test for one vertex of each object against the other. This test is explained in details in Sect. 2.5.13.

2.5.2 Computing Hierarchy-Self Intersections

The algorithm for self-intersecting a hierarchy is very similar to the one presented in Sect. 2.5.1 for intersecting different hierarchies. We still maintain a priority queue of node pairs, but in this case the nodes belong to the same hierarchy. The priority queue is initialized with the root node testing against itself. The following summarizes the recursive process used to self-intersect a hierarchy.

1. Get (and remove) the first element in the queue. Let $n_1 \in H_1$ and $n_2 \in H_1$ be the nodes corresponding to this element.
2. If $n_1 \neq n_2$, then check if the nodes intersect. If they don't intersect, then discard this pair and move on to the next element in the queue.
3. Check if both nodes are leaves of the tree. If they are leaves, then add their primitives to the list of collision candidates. Discard this pair and move on to the next element in the queue.
4. Check if at least one of the nodes is a leaf. If one of the nodes is in fact a leaf, say n_1, then do the following:

(a) Add the pair $(n_1, (n_2)_{left})$ to the priority queue, where $(n_2)_{left}$ is the child to the left of the node n_2.
(b) Add the pair $(n_1, (n_2)_{right})$ to the priority queue, where $(n_2)_{right}$ is the child to the right of the node n_2.

5. At this point, both n_1 and n_2 are internal nodes. The following pairs are added to the priority queue for further intersection tests:
   - $((n_1)_{left}, (n_2)_{left})$,
   - $((n_1)_{left}, (n_2)_{right})$,
   - $((n_1)_{right}, (n_2)_{right})$,
   - if $n_1 \neq n_2$ then add the pair $((n_1)_{right}, (n_2)_{left})$.

The main difference between this algorithm and the one presented in Sect. 2.5.1 is the avoidance of redundant calculations by making sure the children nodes are different before adding them to the priority queue. Notice that

$$\big((n_1)_{left}, (n_2)_{right}\big) = \big((n_1)_{right}, (n_2)_{left}\big)$$

whenever $n_1 = n_2$. At the end, we have a list of collision candidates containing pairs of primitives that need to be tested for intersections. The hierarchy doesn't self-intersect if this list is empty, or no pairs of primitives intersect.

2.5.3 Computing Box–Box Intersections

The intersection test between two boxes is based on the separating-axis theorem. This theorem states that two boxes A and B are disjoint if and only if there exists a separating plane such that the boxes are located on different sides of the plane.

Let $\vec{n}$ be the normal of a plane P, and let d be its nonnegative distance to the origin. The plane P is a separating plane of boxes A and B if

$$\vec{n} \cdot \vec{a} + d \leq 0, \quad \forall \vec{a} \in A \tag{2.8}$$

and

$$\vec{n} \cdot \vec{b} + d > 0, \quad \forall \vec{b} \in B, \tag{2.9}$$

that is, if the projections of A and B along the normal fall on opposite sides of the plane. Equations (2.8) and (2.9) can then be combined into the single equation

$$\vec{n} \cdot \vec{a} < \vec{n} \cdot \vec{b}, \quad \forall \vec{a} \in A, \ \forall \vec{b} \in B. \tag{2.10}$$

Equation (2.10) states that, if P is a separating plane of boxes A and B, then their images are disjoint under axial projection along an axis parallel to the plane normal $\vec{n}$. In other words, $\vec{n}$ is a separating axis of A and B. It can be shown that the separating-axis candidates are the normals to the faces of A and B, and the normals to the planes defined by one edge of A and one edge of B. This results in

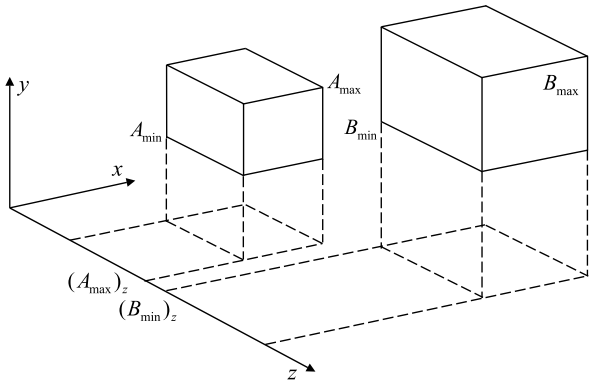

Fig. 2.22 Axis-aligned box–box intersection test. Each box is defined by its minimum and maximum vertices. The intersection test is then carried out by checking whether their projections along each coordinate axis overlap. In this case, the z-axis is a separating axis and the objects do not overlap

15 potential cases to be tested: 3 different face normals for each box plus 9 pair-wise combinations of edges. If none of the potential separating axes actually separates the boxes, then the boxes are guaranteed to be overlapping.

Let us first consider the simple intersection case, where the boxes are aligned with each other and are parallel to the world-coordinate frame's axis. This case occurs in the hierarchical representation of the simulated world, where all boxes in the uniform or multi-level grids have the same orientation with respect to the world-coordinate frame. In such situations, the 15 potential cases are reduced to just 3, since the face normals of each box are the same and the pair-wise combination of their edges always gives another edge. Therefore, the three separating-axis candidates are the axes of the world-coordinate frame.

Let each box be represented by its minimum and maximum vertices, as indicated in Fig. 2.22.

Let $[(A_{min})_i, (A_{max})_i]$ and $[(B_{min})_i, (B_{max})_i]$ be the projections of boxes A and B along the coordinate axis i, for $i = \{x, y, z\}$. The boxes A and B will *not* overlap if and only if

$$\big((A_{max})_i < (B_{min})_i\big) \cup \big((B_{max})_i < (A_{min})_i\big) \tag{2.11}$$

for at least one projection axis $i \in \{x, y, z\}$. This projection axis is then the separating axis for the boxes. On the other hand, if Eq. (2.11) is not satisfied for all projection axes, then the boxes are guaranteed to be overlapping.

Having considered the simple axis-aligned case, let's move on to the more complex case wherein the boxes are arbitrarily oriented with respect to each other. This happens when checking for intersections between boxes in the AABB or OBB hierarchical representations, since they usually have different orientations in the world-coordinate frame.

Let $\vec{T}_A$ and $\mathbf{R_A}$ be the translation vector and rotation matrix from A's local-coordinate frame to the world-coordinate frame. The axis of A in the world-coordinate frame will then be given by the columns of $\mathbf{R_A}$, namely $(\vec{R}_A)_x$, $(\vec{R}_A)_y$ and $(\vec{R}_A)_z$, that is

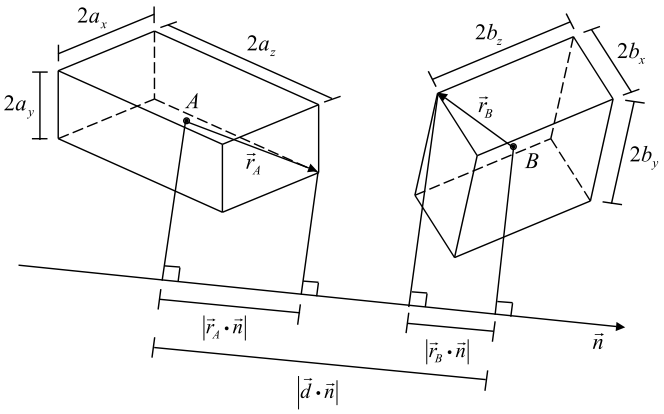

Fig. 2.23 Arbitrarily oriented box–box test. The boxes will not intersect if the axial projection of the distance between their centers is greater than the sum of the axial projection of their half-sides. There are 15 possible axial directions to be tested

$$\mathbf{R_A} = \left((\vec{R}_A)_x \mid (\vec{R}_A)_y \mid (\vec{R}_A)_z \right)$$
$$= \left(\begin{array}{c|c|c} (R_A)_{xx} & R_A)_{yx} & (R_A)_{zx} \\ (R_A)_{xy} & (R_A)_{yy} & (R_A)_{zy} \\ (R_A)_{xz} & (R_A)_{yz} & (R_A)_{zz} \end{array} \right)$$

Similarly, let $\vec{T}_B$ and $\mathbf{R_B}$ be the translation vector and rotation matrix from B's local-coordinate frame to the world-coordinate frame, and the axis of B in the world-coordinate frame be $(\vec{R}_B)_x$, $(\vec{R}_B)_y$ and $(\vec{R}_B)_z$. Let $\vec{d}$ be the distance vector between the center of the boxes in the world-coordinate frame. The boxes A and B will be disjoint if and only if the sum of the projections of their half-sides along the candidate separating axis $\vec{n}$ is less than the projection of their distance vector $\vec{d}$ along $\vec{n}$, that is

$$|\vec{r}_A \cdot \vec{n}| + |\vec{r}_B \cdot \vec{n}| < |\vec{d} \cdot \vec{n}|, \tag{2.12}$$

where $\vec{r}_A$ and $\vec{r}_B$ are the sum of the projections of the half-sides of A and B, respectively. Figure 2.23 illustrates this.

The distances between boxes A and B to the origin of the world-coordinate frame are given by $\vec{T}_A$ and $\vec{T}_B$. Therefore, their distance vector can be directly obtained from

$$\vec{d} = \vec{T}_B - \vec{T}_A. \tag{2.13}$$

The half-sides of each box can be computed from the boxes' minimum and maximum vertices transformed to the world-coordinate frame. Let a_x, a_y and a_z be the half-sides of box A along its axes $(\vec{R}_A)_x$, $(\vec{R}_A)_y$ and $(\vec{R}_A)_z$. Similarly, let b_x, b_y and

b_z be the half-sides of box B along its axes $(\vec{R}_B)_x$, $(\vec{R}_B)_y$ and $(\vec{R}_B)_z$. The sum of the projections of the half-sides of A and B along $\vec{n}$ are then

$$\begin{aligned}\vec{r}_A \cdot \vec{n} &= a_x\left|(\vec{R}_A)_x \cdot \vec{n}\right| + a_y\left|(\vec{R}_A)_y \cdot \vec{n}\right| + a_z\left|(\vec{R}_A)_z \cdot \vec{n}\right| \\ \vec{r}_B \cdot \vec{n} &= b_x\left|(\vec{R}_B)_x \cdot \vec{n}\right| + b_y\left|(\vec{R}_B)_y \cdot \vec{n}\right| + b_z\left|(\vec{R}_B)_z \cdot \vec{n}\right|.\end{aligned} \tag{2.14}$$

Substituting Eqs. (2.14) and (2.13) into (2.12), we have that $\vec{n}$ is a separating axis if and only if

$$\begin{aligned}\left|(\vec{T}_B - \vec{T}_A) \cdot \vec{n}\right| > \big(&a_x\left|(\vec{R}_A)_x \cdot \vec{n}\right| \\ &+ a_y\left|(\vec{R}_A)_y \cdot \vec{n}\right| \\ &+ a_z\left|(\vec{R}_A)_z \cdot \vec{n}\right| \\ &+ b_x\left|(\vec{R}_B)_x \cdot \vec{n}\right| \\ &+ b_y\left|(\vec{R}_B)_y \cdot \vec{n}\right| \\ &+ b_z\left|(\vec{R}_B)_z \cdot \vec{n}\right|\big)\end{aligned} \tag{2.15}$$

is satisfied for the 15 possible combinations of $\vec{n}$, namely $\vec{n} = (\vec{R}_A)_i$, $\vec{n} = (\vec{R}_B)_i$ or $\vec{n} = (\vec{R}_A)_i \times (\vec{R}_B)_j$ for $i, j \in \{x, y, z\}$ and $i \neq j$.

Equation (2.15) can be simplified if we carry out the computations in A's local-coordinate frame, as opposed to the world-coordinate frame. This can be done by translating all points by $-\vec{T}_A$ and rotating them by $\mathbf{R_A}^{-1} = \mathbf{R_A}^t$. This yields

$$\begin{aligned}\vec{T}_A &= (\vec{T}_A - \vec{T}_A) = (0, 0, 0) \\ \mathbf{R_A} &= \mathbf{R_A}^t \mathbf{R_A} = \mathbf{I_3} \\ \vec{T}_B &= \mathbf{R_A}^t (\vec{T}_B - \vec{T}_A) \\ \mathbf{R_B} &= \mathbf{R_A}^t \mathbf{R_B},\end{aligned} \tag{2.16}$$

where $\mathbf{I_3}$ is the 3×3 identity matrix. Substituting Eq. (2.16) into (2.15), we can explicitly derive the equations for all 15 possible tests for finding a separating axis for boxes A and B with respect to A's local-coordinate frame. These results are summarized in Table 2.1.

2.5.4 Computing Sphere–Sphere Intersections

The sphere–sphere intersection test is by far the simplest in this chapter. Two spheres are *not* intersecting if and only if the distance between their centers is greater than the sum of their radii. This is illustrated by Fig. 2.24.

Let r_A and $\vec{c}_A$ be the radius and center of sphere A, respectively. Similarly, let r_B and $\vec{c}_B$ be the radius and center of sphere B. The spheres will not overlap if and only if

$$|\vec{c}_A - \vec{c}_B| > (r_A + r_B).$$

Table 2.1 The 15 candidate separating axes and their associated tests with respect to A's local-coordinate frame. The boxes are overlapping if and only if all tests fail

Separating axis $\vec{n}$	Simplified overlap test
$(\vec{R}_A)_x$	$\lvert(T_B)_x\rvert > (a_x + b_x\lvert(R_B)_{xx}\rvert + b_y\lvert(R_B)_{xy}\rvert + b_z\lvert(R_B)_{xz}\rvert)$
$(\vec{R}_A)_y$	$\lvert(T_B)_y\rvert > (a_y + b_x\lvert(R_B)_{yx}\rvert + b_y\lvert(R_B)_{yy}\rvert + b_z\lvert(R_B)_{yz}\rvert)$
$(\vec{R}_A)_z$	$\lvert(T_B)_z\rvert > (a_z + b_x\lvert(R_B)_{zx}\rvert + b_y\lvert(R_B)_{zy}\rvert + b_z\lvert(R_B)_{zz}\rvert)$
$(\vec{R}_B)_x$	$\lvert(T_B)_x(R_B)_{xx} + (T_B)_y(R_B)_{yx} + (T_B)_z(R_B)_{zx}\rvert$ $> (b_x + a_x\lvert(R_B)_{xx}\rvert + a_y\lvert(R_B)_{yx}\rvert + a_z\lvert(R_B)_{zx}\rvert)$
$(\vec{R}_B)_y$	$\lvert(T_B)_x(R_B)_{xy} + (T_B)_y(R_B)_{yy} + (T_B)_z(R_B)_{zy}\rvert$ $> (b_y + a_x\lvert(R_B)_{xy}\rvert + a_y\lvert(R_B)_{yy}\rvert + a_z\lvert(R_B)_{zy}\rvert)$
$(\vec{R}_B)_z$	$\lvert(T_B)_x(R_B)_{xz} + (T_B)_y(R_B)_{yz} + (T_B)_z(R_B)_{zz}\rvert$ $> (b_z + a_x\lvert(R_B)_{xz}\rvert + a_y\lvert(R_B)_{yz}\rvert + a_z\lvert(R_B)_{zz}\rvert)$
$(\vec{R}_A)_x \times (\vec{R}_B)_x$	$\lvert(T_B)_z(R_B)_{yx} - (T_B)_y(R_B)_{zx}\rvert$ $> (a_y\lvert(R_B)_{zx}\rvert + a_z\lvert(R_B)_{yx}\rvert + b_y\lvert(R_B)_{xz}\rvert + b_z\lvert(R_B)_{xy}\rvert)$
$(\vec{R}_A)_x \times (\vec{R}_B)_y$	$\lvert(T_B)_z(R_B)_{yy} - (T_B)_y(R_B)_{zy}\rvert$ $> (a_y\lvert(R_B)_{zy}\rvert + a_z\lvert(R_B)_{yy}\rvert + b_x\lvert(R_B)_{xz}\rvert + b_z\lvert(R_B)_{xx}\rvert)$
$(\vec{R}_A)_x \times (\vec{R}_B)_z$	$\lvert(T_B)_z(R_B)_{yz} - (T_B)_y(R_B)_{zz}\rvert$ $> (a_y\lvert(R_B)_{zz}\rvert + a_z\lvert(R_B)_{yz}\rvert + b_x\lvert(R_B)_{xy}\rvert + b_y\lvert(R_B)_{xx}\rvert)$
$(\vec{R}_A)_y \times (\vec{R}_B)_x$	$\lvert(T_B)_x(R_B)_{zx} - (T_B)_z(R_B)_{xx}\rvert$ $> (a_x\lvert(R_B)_{zx}\rvert + a_z\lvert(R_B)_{xx}\rvert + b_y\lvert(R_B)_{yz}\rvert + b_z\lvert(R_B)_{yy}\rvert)$
$(\vec{R}_A)_y \times (\vec{R}_B)_y$	$\lvert(T_B)_x(R_B)_{zy} - (T_B)_z(R_B)_{xy}\rvert$ $> (a_x\lvert(R_B)_{zy}\rvert + a_z\lvert(R_B)_{xy}\rvert + b_x\lvert(R_B)_{yz}\rvert + b_z\lvert(R_B)_{yx}\rvert)$
$(\vec{R}_A)_y \times (\vec{R}_B)_z$	$\lvert(T_B)_x(R_B)_{zz} - (T_B)_z(R_B)_{xz}\rvert$ $> (a_x\lvert(R_B)_{zz}\rvert + a_z\lvert(R_B)_{xz}\rvert + b_x\lvert(R_B)_{yy}\rvert + b_y\lvert(R_B)_{yx}\rvert)$
$(\vec{R}_A)_z \times (\vec{R}_B)_x$	$\lvert(T_B)_y(R_B)_{xx} - (T_B)_x(R_B)_{yx}\rvert$ $> (a_x\lvert(R_B)_{yx}\rvert + a_y\lvert(R_B)_{xx}\rvert + b_y\lvert(R_B)_{zz}\rvert + b_z\lvert(R_B)_{zy}\rvert)$
$(\vec{R}_A)_z \times (\vec{R}_B)_y$	$\lvert(T_B)_y(R_B)_{xy} - (T_B)_x(R_B)_{yy}\rvert$ $> (a_x\lvert(R_B)_{yy}\rvert + a_y\lvert(R_B)_{xy}\rvert + b_x\lvert(R_B)_{zz}\rvert + b_z\lvert(R_B)_{zx}\rvert)$
$(\vec{R}_A)_z \times (\vec{R}_B)_z$	$\lvert(T_B)_y(R_B)_{xz} - (T_B)_x(R_B)_{yz}\rvert$ $> (a_x\lvert(R_B)_{yz}\rvert + a_y\lvert(R_B)_{xz}\rvert + b_x\lvert(R_B)_{zy}\rvert + b_y\lvert(R_B)_{zx}\rvert)$

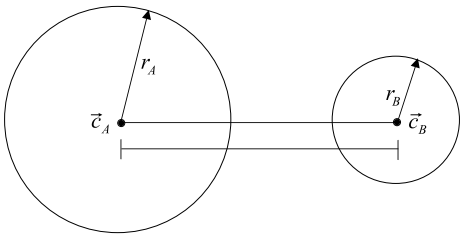

Fig. 2.24 The sphere–sphere intersection test can be quickly conducted by comparing the distance between the centers of the sphere with the sum of their radii

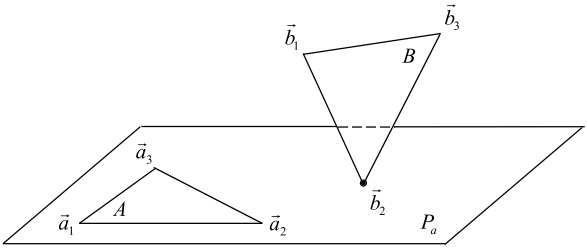

Fig. 2.25 Case where just one vertex of triangle B touches the plane P_a containing triangle A. As shown, $\vec{b}_2$ lies outside A and the triangles do not intersect

2.5.5 Computing Triangle–Triangle Intersections

The triangle–triangle intersection test is considered a primitive–primitive intersection test, since the triangles are in fact faces of the objects in the simulation. Let triangles A and B be defined by vertices $\vec{a}_1$, $\vec{a}_2$, $\vec{a}_3$ and $\vec{b}_1$, $\vec{b}_2$ and $\vec{b}_3$, respectively. The first step of the intersection test is to conduct a quick rejection test. This test consists of determining whether all vertices of one triangle lie on the same side of the plane defined by the other triangle. Let P_a and P_b be the planes defined by triangles A and B, respectively. Let $\vec{n}_a$ and $\vec{n}_b$ be the normal vectors of P_a and P_b. The normals can be directly computed from the vertex list as

$$\vec{n}_a = (\vec{a}_2 - \vec{a}_1) \times (\vec{a}_3 - \vec{a}_2)$$
$$\vec{n}_b = (\vec{b}_2 - \vec{b}_1) \times (\vec{b}_3 - \vec{b}_2).$$

The vertices of triangle B will lie on the same side of P_a if and only if

$$\begin{aligned} &\vec{n}_a \cdot (\vec{b}_1 - \vec{a}_1) \\ &\vec{n}_a \cdot (\vec{b}_2 - \vec{a}_1) \\ &\vec{n}_a \cdot (\vec{b}_3 - \vec{a}_1) \end{aligned} \tag{2.17}$$

are not zero and have the same sign. If they do not have the same sign, then the following cases can occur.

Case 1 Two of the three equations defined in (2.17) have the same sign and the third evaluates to zero, say that corresponding to $\vec{b}_2$. In this case, the intersection between triangle B and plane P_a is single point, that is, vertex $\vec{b}_2$ (see Fig. 2.25). The triangle–triangle intersection test is then reduced to check whether $\vec{b}_2$ lies inside triangle A. This point-in-triangle test can be quickly done by considering the line segment connecting $\vec{b}_2$ to the barycenter of A. If this line segment intersects one of the edges of triangle A, then $\vec{b}_2$ lies outside the triangle. Otherwise, $\vec{b}_2$ lies inside triangle A, and triangle B intersects triangle A. More details on how to implement the point-in-triangle test are given in Sect. 2.5.12.

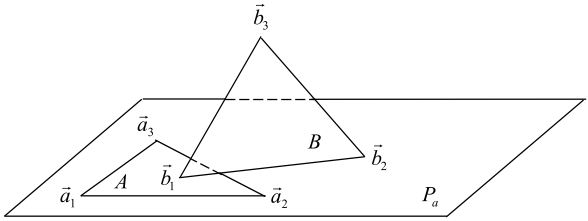

Fig. 2.26 Case where an edge of triangle B is coplanar with triangle A. As shown, vertices $\vec{b}_1$ and $\vec{b}_2$ lie inside and outside A, respectively, and the triangles intersect

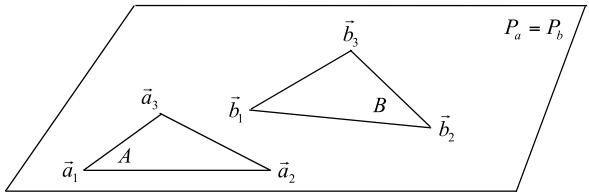

Fig. 2.27 Case where the triangles A and B are coplanar. As shown, the vertices of B lie outside A, and the triangles do not intersect

Case 2 Two of the three equations defined in (2.17) evaluate to zero, say those corresponding to $\vec{b}_1$ and $\vec{b}_2$. In this case, the intersection between triangle B and plane P_a is the line segment defined by $(\vec{b}_1, \vec{b}_2)$ (see Fig. 2.26). We can then apply the edge–edge intersection test given in Sect. 2.5.12, to check for intersections between line segment $(\vec{b}_1, \vec{b}_2)$ and each one of the triangle edges $(\vec{a}_1, \vec{a}_2)$, $(\vec{a}_2, \vec{a}_3)$ and $(\vec{a}_3, \vec{a}_1)$. An intersection is detected whenever $\vec{b}_1$ or $\vec{b}_2$ lie inside triangle A. If no intersections are detected, then we still need to check if segment $(\vec{b}_1, \vec{b}_2)$ lies completely inside triangle A. This requires an additional point-in-triangle test for $\vec{b}_1$ or $\vec{b}_2$.

Case 3 All three equations defined in (2.17) evaluate to zero. In this case, triangles A and B are coplanar (see Fig. 2.27). The intersection test can then be reduced to nine edge–edge tests by considering the three edges of triangle B checked against the three edges of triangle A. If at least one of the B edges intersect an edge of A, then the triangles intersect. However, if all edge–edge intersection tests fail, then there is still the possibility that A is completely inside B, or vice-versa. Therefore, an extra two point-in-triangle tests are performed: one for a vertex of A against B, and another for a vertex of B against A.

Case 4 None of the three equations defined in (2.17) evaluates to zero. In this case, we shall have two vertices of B on one side of plane P_a, and the third vertex on the other side of P_a. This is illustrated by Fig. 2.28.

Let $\vec{b}_2$ be the vertex that lies on the opposite side of plane P_a. Triangle B will then intersect plane P_a in two points $\vec{p}_1$ and $\vec{p}_2$ defining a line segment on the plane of A. These points can be computed as follows. Consider edge $e_1 = (\vec{b}_1, \vec{b}_2)$ given by its parameterized equation

$$\vec{p} = \vec{b}_1 + t(\vec{b}_2 - \vec{b}_1), \tag{2.18}$$

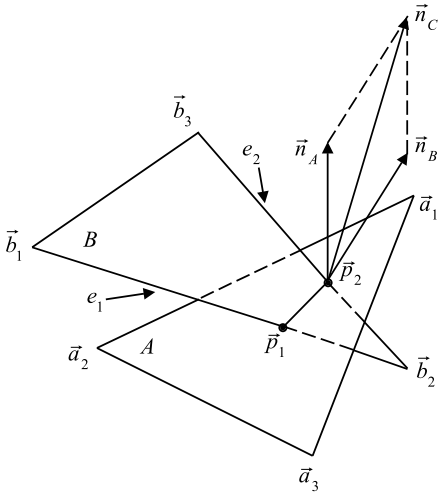

Fig. 2.28 Triangle–triangle intersection test for the case where no vertices of triangle B are coplanar with triangle A. One vertex of B will lie on the opposite side of the other two vertices, with respect to the plane defined by triangle A

where $0 \leq t \leq 1$ and $\vec{p}$ is a point on the edge. The plane equation of P_a is given by

$$\vec{n}_a \cdot \vec{p} = d, \tag{2.19}$$

where $\vec{p}$ is any point on the plane and d is the plane constant given by

$$d = \vec{n}_a \cdot \vec{a}_1 = \vec{n}_a \cdot \vec{a}_2 = \vec{n}_a \cdot \vec{a}_3.$$

Edge e_1 intersects plane P_a at a point $\vec{p}_1$ that satisfies both Eqs. (2.18) and (2.19). Substituting Eq. (2.18) into (2.19), we can compute the value of t corresponding to the intersection point $\vec{p}_1$, that is

$$t_p = \frac{d - \vec{n}_a \cdot \vec{b}_1}{\vec{n}_a \cdot (\vec{b}_2 - \vec{b}_1)}. \tag{2.20}$$

Substituting Eq. (2.20) back into (2.18), we can immediately find the intersection point $\vec{p}_1$. A similar computation can be done to find the intersection point $\vec{p}_2$ between edge $e_2 = (\vec{b}_2, \vec{b}_3)$ and plane P_a.

The line segment $(\vec{p}_1, \vec{p}_2)$ can then be checked for intersection with triangle A. We can apply the edge–edge intersection test between $(\vec{p}_1, \vec{p}_2)$ and the edges of triangle A. If an intersection is detected, then triangle B intersects triangle A. Otherwise, we still need to test if edge $(\vec{p}_1, \vec{p}_2)$ lies completely inside triangle A. This requires an additional point-in-triangle test for $\vec{p}_1$ or $\vec{p}_2$.

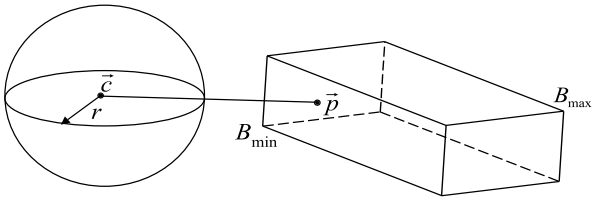

Fig. 2.29 The closest point to the sphere is on the boundary of the box and minimizes the distance to the center given by Eq. (2.21)

2.5.6 Computing Box–Sphere Intersections

The intersection between an axis-aligned box and a sphere is carried out by considering the point in the boundary of the box that is closest to the sphere, and checking whether its distance to the center of the sphere is greater than the sphere's radius. If the distance is less than or equal to the sphere's radius, then the box intersects the sphere. If the box is oriented, then this intersection test can be carried out on its local-coordinate frame, intersecting the sphere transformed to local space with the local-space axis-aligned box representation of the oriented box.

Let $\vec{p}$ be a point in the box, and let $\vec{c}$ and r be the sphere's center and radius, respectively. Let x_{min}, x_{max}, y_{min}, y_{max}, z_{min} and z_{max} define the minimum and maximum values of the boundary of the box along each of the coordinated axes, as shown in Fig. 2.29.

The square of the distance from $\vec{p}$ to $\vec{c}$ is then given by

$$d^2 = (c_x - p_x)^2 + (c_y - p_y)^2 + (c_z - p_z)^2. \tag{2.21}$$

The point $\vec{p}$ that is closest to the sphere is that which minimizes Eq. (2.21), subject to the following constraints:

$$\begin{aligned} x_{min} \leq p_x \leq x_{max} \\ y_{min} \leq p_y \leq y_{max} \\ z_{min} \leq p_z \leq z_{max}. \end{aligned}$$

Notice that each term of Eq. (2.21) is nonnegative and can be independently minimized. For example, if $x_{min} \leq c_x \leq x_{max}$, then $p_x = c_x$ minimizes the term $(c_x - p_x)^2$. However, if $c_x < x_{min}$ or $c_x > x_{max}$, then $p_x = x_{min}$ or $p_x = x_{max}$ minimizes the term, respectively. We do a similar analysis for finding the value of p_y and p_z that minimizes their corresponding quadratic terms.

Having determined the coordinates of the closest point to the sphere, we just need to compare its distance to the center of the sphere with the sphere's radius by substituting the coordinates of $\vec{p}$ into Eq. (2.21), and checking whether

$$d^2 \leq r^2. \tag{2.22}$$

The box will intersect the sphere if and only if Eq. (2.22) is satisfied.

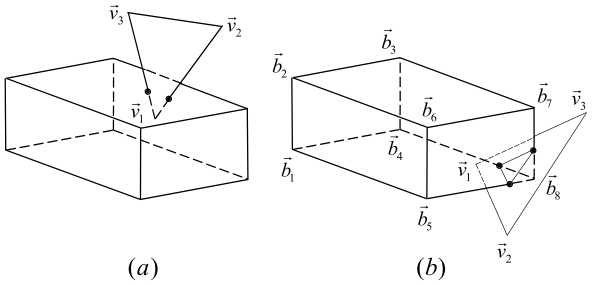

Fig. 2.30 (**a**) The triangle intersects the box whenever one of its vertices lies inside the box; (**b**) The plane containing the triangle intersects the box. Edges of the box that have vertices at opposite sides of the plane, in this case edges $\overline{b_5 b_8}$, $\overline{b_4 b_8}$ and $\overline{b_7 b_8}$, need to be checked for intersection with the triangle

2.5.7 Computing Box–Triangle Intersections

The box–triangle intersection test can be quickly carried out in at most three steps. In the first step, we check whether the vertices of the triangle are inside the box. If at least one of the vertices is inside the box, then the triangle intersects the box.

Let the box be defined by its minimum and maximum vertices, and let $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$ be the vertices of the triangle (see Fig. 2.30(a)). Vertex $\vec{v}_i$ is inside the box if and only if

$$\begin{aligned} x_{min} \leq (v_i)_x \leq x_{max} \\ y_{min} \leq (v_i)_y \leq y_{max} \\ z_{min} \leq (v_i)_z \leq z_{max}. \end{aligned} \tag{2.23}$$

If at least one of the vertices $\vec{v}_1$, $\vec{v}_2$ or $\vec{v}_3$ satisfies Eq. (2.23), then the triangle intersects the box. Otherwise, we proceed to the second step.

In the second step, we check whether the plane containing the triangle intersects the box. This can be done by checking whether the eight vertices of the box lie on the same side of the plane (see Fig. 2.30(b)). Let $\vec{n}$ be the normal of the triangle and d the plane constant determined from

$$d = \vec{n} \cdot \vec{v}_i, \quad \text{for } i \in \{1, 2, 3\}.$$

A point $\vec{p}$ is classified with respect to the plane containing the triangle as follows:

$$\begin{aligned} &\text{If } \vec{n} \cdot \vec{p} - d > 0 \quad \Rightarrow \quad \vec{p} \text{ is on positive half-plane} \\ &\text{If } \vec{n} \cdot \vec{p} - d = 0 \quad \Rightarrow \quad \vec{p} \text{ lies on the plane} \\ &\text{If } \vec{n} \cdot \vec{p} - d < 0 \quad \Rightarrow \quad \vec{p} \text{ is on negative half-plane.} \end{aligned} \tag{2.24}$$

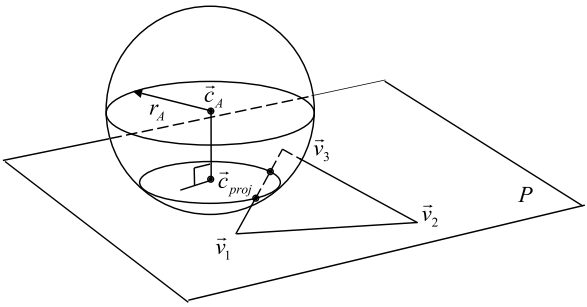

Fig. 2.31 The triangle intersects the sphere whenever one of its vertices lies inside the sphere, that is, the distance from one of its vertices to the center of the sphere is less than or equal to the sphere's radius

Using Eq. (2.24), we classify each vertex of the box according to its relative position with respect to the plane. If all vertices lie on the same half-space, then we can immediately conclude that the box does not intersect the triangle. Otherwise, we need to consider the edges of the box that intersect the plane, that is, the edges that have vertices at opposite sides of the plane, or one vertex on the plane and another on either side. These edges define line segments and a line segment–triangle intersection test is done for each of them, as explained in detail in Sect. 2.5.10.

2.5.8 Computing Sphere–Triangle Intersections

The sphere–triangle test is more complex than the box–triangle test, in the sense that it has more steps to be carried out before we can determine whether the sphere is intersecting the triangle.

The first step is to check whether the plane that contains the triangle intersects the sphere. This can be done by comparing the distance of the plane to the center of the sphere with the radius of the sphere. Let r_A and $\vec{c}_A$ be the sphere's radius and center. Let $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$ be the vertices of the triangle defining plane P (see Fig. 2.31). Let $\vec{n}$ and d_n be the plane normal and plane constant. The distance between the plane P and the sphere's center is then

$$d_A = |\vec{n} \cdot \vec{c}_A - d_n|.$$

The plane containing the triangle intersects the sphere whenever

$$d_A \leq r_A.$$

If this is the case, then we proceed with the sphere–triangle intersection test by checking whether the vertices of the triangle are inside the sphere. If at least one of the vertices is inside the sphere, then the triangle intersects the sphere. Let d_i be the distance between vertex $\vec{v}_i$ and the sphere's center, that is

$$d_i = |\vec{v}_i - \vec{c}_A|.$$

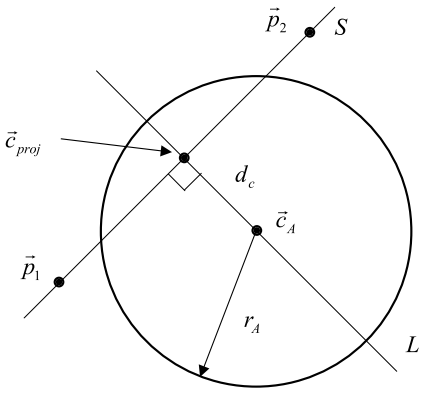

Fig. 2.32 Intersection test between a sphere and a line segment. We only need consider the intersection of the line segment with the circle resulting from the intersection of the sphere and the plane defined by the center of the sphere, and the end points of the line segment

The sphere will intersect the triangle if

$$d_i \leq r_A \tag{2.25}$$

for at least one vertex $\vec{v}_i$ (see Fig. 2.31). If this is not the case, then we proceed to the third step of the sphere–triangle intersection test. In this step, we project the sphere onto the plane containing the triangle, and check whether the projected center lies inside the triangle. The projected center $\vec{c}_{proj}$ is determined from

$$\vec{c}_{proj} = \vec{c}_A - d_n \vec{n}.$$

We can use the point-in-triangle test already explained in Sect. 2.5.12 to see whether the projected center lies inside the triangle. If the projected center $\vec{c}_{proj}$ lies inside the triangle, then the sphere intersects the triangle (see Fig. 2.31). Otherwise, we need to do one more test to check whether the triangle edges intersect the sphere, as explained in the next section.

2.5.9 Computing Line Segment–Sphere Intersections

Let $\vec{p}_1$ and $\vec{p}_2$ define a line segment S, and $\vec{c}_A$ and r_A be the sphere's center and radius, respectively. Consider the line L passing through $\vec{c}_A$ and perpendicular to the line segment S (see Fig. 2.32).

Line L will intersect the line segment S at a point $\vec{c}_{proj}$ such that

$$\vec{c}_{proj} = \vec{p}_1 + t(\vec{p}_2 - \vec{p}_1)$$

for t, given by

$$t = \frac{(\vec{p}_2 - \vec{p}_1) \cdot \vec{c}_A - (\vec{p}_2 - \vec{p}_1) \cdot \vec{p}_1}{(\vec{p}_2 - \vec{p}_1) \cdot (\vec{p}_2 - \vec{p}_1)},$$

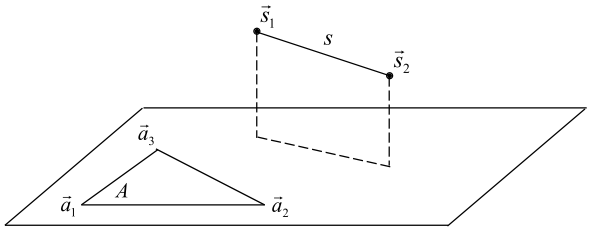

Fig. 2.33 Case where the vertices defining the line segment lie on the same side of the plane containing triangle A. As shown, the line segment does not intersect the triangle

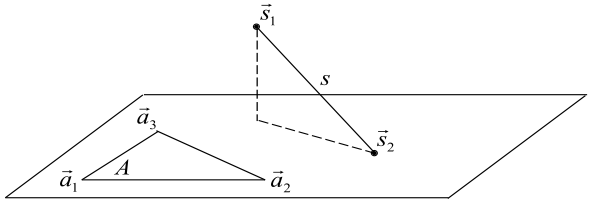

Fig. 2.34 Case where one vertex of the line segment is coplanar with triangle A. As shown, vertex $\vec{s}_2$ lies outside A, and the line segment does not intersect the triangle

that is, $\vec{c}_{proj}$ is the projection of the center of the sphere onto the line segment's supporting line. The distance between the projection point $\vec{c}_{proj}$ and the center of the sphere is directly obtained from

$$d_c^2 = (\vec{c}_{proj} - \vec{c}_A) \cdot (\vec{c}_{proj} - \vec{c}_A).$$

Having determined the projection point $\vec{c}_{proj}$ and its (squared) distance d_c^2 to the center of the sphere, one of the following three cases occurs.

1. If $d_c^2 > r_A^2$, then the sphere does not intersect the triangle.
2. If $d_c^2 = r_A^2$, then the supporting line is tangential to the sphere. If the projection point $\vec{c}_{proj}$ is inside the segment, that is, if $0 \leq t \leq 1$, then the line segment intersects the sphere.
3. If $d_c^2 < r_A^2$, then the supporting line intersects the sphere. The segment intersects the sphere if the projection point $\vec{c}_{proj}$ is inside the segment. Otherwise, an intersection occurs if the closest end point to $\vec{c}_{proj}$ is inside the sphere. The closest end point is $\vec{p}_1$ if $t \leq 0$, or $\vec{p}_2$ if $t \geq 1$.

2.5.10 Computing Line Segment–Triangle Intersections

The intersection of a line segment S with a triangle A can be viewed as a subset of the intersection test between two triangles. Let the line segment be defined by vertices $\vec{s}_1$ and $\vec{s}_2$, and the triangle be defined by vertices $\vec{a}_1$, $\vec{a}_2$ and $\vec{a}_3$.

First, we check whether the vertices defining the line segment lie on the same side of the plane containing the triangle (see Fig. 2.33). If this is so, then we can quickly conclude that the segment does not intersect the triangle. Otherwise, we can have one of the following three cases.

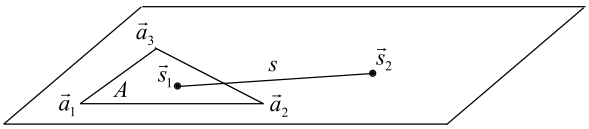

Fig. 2.35 Case where the line segment is coplanar with triangle A. In the situation shown, vertices $\vec{s}_1$ and $\vec{s}_2$ lie inside and outside A, respectively, and the line segment does intersect the triangle

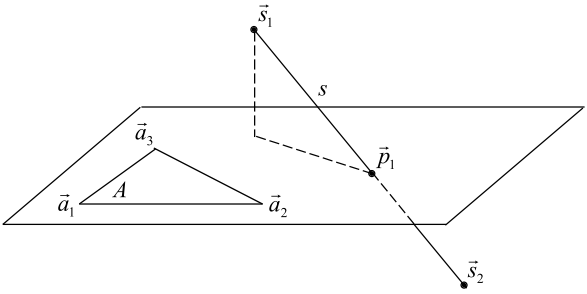

Fig. 2.36 Case where the vertices defining the line segment lie on opposite sides of the plane containing triangle A. As shown, the intersection point $\vec{p}_1$ between the line segment and the plane lies outside A, and the line segment does not intersect the triangle

Case 1 One vertex of the line segment, say vertex $\vec{s}_2$, lies on the plane that contains the triangle, and the other lies on either side (see Fig. 2.34). In this case, we use the point-in-triangle test for checking whether $\vec{s}_2$ lies inside A. The line segment intersects the triangle only if $\vec{s}_2$ lies inside A.

Case 2 Both vertices of the line segment lie on the plane containing A (see Fig. 2.35). Again, we use the edge–edge test for checking for intersections between the line segment and the edges of A. The line segment intersects the triangle only if it crosses one of its edges, or if both vertices lie inside the triangle.

Case 3 The vertices of the line segment lie on opposite sides of the plane containing the triangle (see Fig. 2.36). Let $\vec{p}_1$ be the intersection between the line segment and the plane containing the triangle. Using the point-in-triangle test, we can check whether $\vec{p}_1$ lies inside the triangle.

2.5.11 Computing Line Segment–Box Intersections

A line segment intersects an axis-aligned box if either one of its vertices lie inside the box, or if it crosses one of the faces defining the box. In the case of oriented box, this same test can be carried out in its local-coordinate frame.

Let the box be defined by its minimum and maximum vertices $\vec{b}_{lower}$ and $\vec{b}_{upper}$, respectively. Let $(\vec{s}_1, \vec{s}_2)$ define the line segment being tested for intersection. As explained in Sect. 2.5.7, the line points $\vec{s}_i$ with $i \in 1, 2$, lie inside the box if they satisfy Eq. (2.23). If this is the case, then the line segment does intersect the box. Otherwise, we need to test whether it intersects one of the box's faces. This intersection test can be optimized if the direction of the line segment is taken into account to identify which three out of the six faces need to be considered. Figure 2.37 illustrates this.

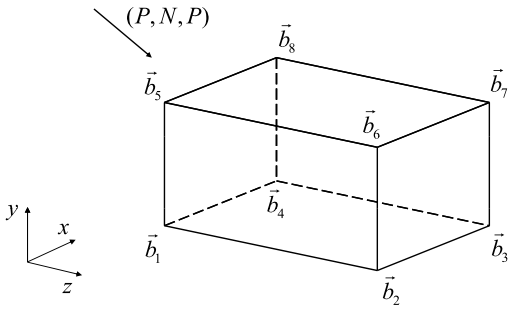

Fig. 2.37 The line segment direction is classified according to the sign of its projections on each coordinate axis. In this case, the line segment has a (P, N, P) direction, that is, positive on X, negative on Y and positive on Z. We only need to test for intersections with three out of six faces, namely $(\vec{b}_1, \vec{b}_4, \vec{b}_8, \vec{b}_5)$, $(\vec{b}_5, \vec{b}_6, \vec{b}_7, \vec{b}_8)$ and $(\vec{b}_1, \vec{b}_2, \vec{b}_6, \vec{b}_5)$

Let $\vec{d}$ be the line segment direction given by

$$\vec{d} = \vec{s}_2 - \vec{s}_1 = (d_x, d_y, d_z).$$

The direction is classified according to the sign of its components d_i as follows:

$$\begin{aligned}
&(d_x < 0), (d_y < 0), (d_z < 0) \rightarrow (N, N, N)\\
&(d_x < 0), (d_y < 0), (d_z \geq 0) \rightarrow (N, N, P)\\
&(d_x < 0), (d_y \geq 0), (d_z < 0) \rightarrow (N, P, N)\\
&(d_x < 0), (d_y \geq 0), (d_z \geq 0) \rightarrow (N, P, P)\\
&(d_x \geq 0), (d_y < 0), (d_z < 0) \rightarrow (P, N, N)\\
&(d_x \geq 0), (d_y < 0), (d_z \geq 0) \rightarrow (P, N, P)\\
&(d_x \geq 0), (d_y \geq 0), (d_z < 0) \rightarrow (P, P, N)\\
&(d_x \geq 0), (d_y \geq 0), (d_z \geq 0) \rightarrow (P, P, P),
\end{aligned}$$

with N and P standing for negative and positive, respectively. A set of three faces is pre-assigned to each different classification, and the line needs to be tested for intersection with each one of them. Following the notation of Fig. 2.37 for the box–vertex assignment, the set of three faces associated with each classification is shown in Table 2.2. The corresponding vertex assignment of each face is summarized in Table 2.3.

The performance of the algorithm can be further improved if we do a quick rejection test before computing the actual intersections between the line segment and the box faces. This quick rejection test considers the silhouette formed by the set of three faces when viewing the box along the line direction, as shown in Fig. 2.38.

If we replace the line segment with a ray, with origin at $\vec{s}_1$ and direction $\vec{d}$, we can test whether the ray passes through the inside region of the silhouette. If the ray fails to pass through the inside of the silhouette, then the line segment does not intersect

Table 2.2 The set of three faces that need to be tested for intersection with the line segment depending on the classification of its direction

Classification	Set of faces
(N, N, N)	Top, Front, Right
(N, N, P)	Top, Back, Right
(N, P, N)	Bottom, Front, Right
(N, P, P)	Bottom, Back, Right
(P, N, N)	Top, Front, Left
(P, N, P)	Top, Back, Left
(P, P, N)	Bottom, Front, Left
(P, P, P)	Bottom, Back, Left

Table 2.3 Face–vertex assignment with face normals pointing outwards

Face label	Face vertices
Top	$(\vec{b}_5, \vec{b}_6, \vec{b}_7, \vec{b}_8)$
Bottom	$(\vec{b}_1, \vec{b}_4, \vec{b}_3, \vec{b}_2)$
Front	$(\vec{b}_2, \vec{b}_3, \vec{b}_7, \vec{b}_6)$
Back	$(\vec{b}_1, \vec{b}_5, \vec{b}_8, \vec{b}_4)$
Left	$(\vec{b}_1, \vec{b}_2, \vec{b}_6, \vec{b}_5)$
Right	$(\vec{b}_3, \vec{b}_4, \vec{b}_8, \vec{b}_7)$

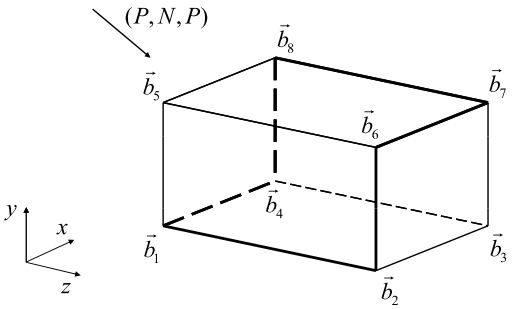

Fig. 2.38 The silhouette of the box is highlighted for the case in which the line direction has a (P, N, P) classification. Notice that the silhouette will always contain exactly six edges of the box

the box. Notice that the silhouette is always made of six edges. If we consider the relative position of the ray with respect to each of these edges, the ray will pass through the silhouette only if it lies on the inside side of each of its edges. The relative orientation of the ray with respect to an edge of the box is obtained from

$$side(ray, edge) = -(\vec{d} \cdot \vec{n}),$$

where $\vec{n}$ is the normal vector of the plane defined by $\vec{s}_1$ (i.e., the ray's origin) and the edge $(\vec{b}_i, \vec{b}_j)$, that is

$$\vec{n} = (\vec{b}_i - \vec{s}_1) \times (\vec{b}_j - \vec{s}_1),$$

as illustrated in Fig. 2.39.

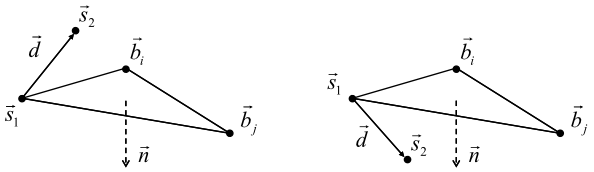

Fig. 2.39 The relative orientation of the ray and the box's edge is determined from the sign of the scalar product between the ray direction $\vec{d}$ and the plane normal $\vec{n}$, defined by points $\vec{s}_1$, $\vec{b}_i$ and $\vec{b}_j$; (**a**) Case in which $\vec{d}$ points away from the normal, resulting in $side(ray, edge) > 0$; (**b**) $\vec{d}$ points along the normal giving $side(ray, edge) < 0$

Table 2.4 shows the side tests that need to be done for each ray classification. The fact that the box is axis-aligned considerably simplifies the final expressions obtained for the $side(ray, edge)$ relation, as shown in Table 2.5.

2.5.12 Point-in-Triangle and Edge–Edge Intersection Tests

All point-in-triangle tests can be transformed into coplanar segment–segment intersection tests. This test can be efficiently implemented as follows. Let $s_1 = (\vec{p}_1, \vec{p}_2)$ and $s_2 = (\vec{q}_1, \vec{q}_2)$ be the line segments being tested for intersection, and let $\vec{n}$ be the normal vector of the plane that contains both segments. The parameterized equations of the segments are then

$$\vec{p} = \vec{p}_1 + t(\vec{p}_2 - \vec{p}_1)$$
$$\vec{q} = \vec{q}_1 + m(\vec{q}_2 - \vec{q}_1),$$

with $0 \leq t \leq 1$ and $0 \leq m \leq 1$. The first step of the intersection test consists of carrying out a quick rejection test. This test consists of checking whether the line segments are parallel, that is, checking whether

$$(\vec{p}_2 - \vec{p}_1) \times (\vec{q}_2 - \vec{q}_1) = \vec{0}.$$

If the line segments are not parallel, they will intersect if and only if there exist $t = t_p$ and $m = m_q$ such that

$$\vec{p}_1 + t_p(\vec{p}_2 - \vec{p}_1) = \vec{q}_1 + m_q(\vec{q}_2 - \vec{q}_1), \tag{2.26}$$

with $0 \leq t_p \leq 1$ and $0 \leq m_q \leq 1$. Equation (2.26) can be solved for t_p and m_q if we consider two auxiliary vectors $\vec{k}_p$ and $\vec{k}_q$ given by

$$\begin{aligned} \vec{k}_p &= \vec{n} \times (\vec{p}_2 - \vec{p}_1) \\ \vec{k}_q &= \vec{n} \times (\vec{q}_2 - \vec{q}_1), \end{aligned} \tag{2.27}$$

Table 2.4 Sign of side relations for each classification. The intersection is discarded if any of the relations is true

Classification	Quick rejection test
(N, N, N)	$(\vec{s}_1)_x < (\vec{b}_{lower})_x \cup (\vec{s}_1)_y < (\vec{b}_{lower})_y \cup (\vec{s}_1)_z < (\vec{b}_{lower})_z$ $\cup\, side(\vec{b}_3, \vec{b}_4) < 0 \cup side(\vec{b}_6, \vec{b}_5) > 0 \cup side(\vec{b}_2, \vec{b}_6) > 0$ $\cup\, side(\vec{b}_4, \vec{b}_8) < 0 \cup side(\vec{b}_8, \vec{b}_5) < 0 \cup side(\vec{b}_3, \vec{b}_2) > 0$
(N, N, P)	$(\vec{s}_1)_x < (\vec{b}_{lower})_x \cup (\vec{s}_1)_y < (\vec{b}_{lower})_y \cup (\vec{s}_1)_z > (\vec{b}_{upper})_z$ $\cup\, side(\vec{b}_3, \vec{b}_4) < 0 \cup side(\vec{b}_6, \vec{b}_5) > 0 \cup side(\vec{b}_3, \vec{b}_7) > 0$ $\cup\, side(\vec{b}_1, \vec{b}_5) < 0 \cup side(\vec{b}_4, \vec{b}_1) < 0 \cup side(\vec{b}_7, \vec{b}_6) > 0$
(N, P, N)	$(\vec{s}_1)_x < (\vec{b}_{lower})_x \cup (\vec{s}_1)_y > (\vec{b}_{upper})_y \cup (\vec{s}_1)_z < (\vec{b}_{lower})_z$ $\cup\, side(\vec{b}_2, \vec{b}_1) < 0 \cup side(\vec{b}_7, \vec{b}_8) > 0 \cup side(\vec{b}_2, \vec{b}_6) > 0$ $\cup\, side(\vec{b}_4, \vec{b}_8) < 0 \cup side(\vec{b}_7, \vec{b}_6) < 0 \cup side(\vec{b}_4, \vec{b}_1) > 0$
(N, P, P)	$(\vec{s}_1)_x < (\vec{b}_{lower})_x \cup (\vec{s}_1)_y > (\vec{b}_{upper})_y \cup (\vec{s}_1)_z > (\vec{b}_{upper})_z$ $\cup\, side(\vec{b}_2, \vec{b}_1) < 0 \cup side(\vec{b}_7, \vec{b}_8) > 0 \cup side(\vec{b}_3, \vec{b}_7) > 0$ $\cup\, side(\vec{b}_1, \vec{b}_5) < 0 \cup side(\vec{b}_3, \vec{b}_2) < 0 \cup side(\vec{b}_8, \vec{b}_5) > 0$
(P, N, N)	$(\vec{s}_1)_x > (\vec{b}_{upper})_x \cup (\vec{s}_1)_y < (\vec{b}_{lower})_y \cup (\vec{s}_1)_z < (\vec{b}_{lower})_z$ $\cup\, side(\vec{b}_7, \vec{b}_8) < 0 \cup side(\vec{b}_2, \vec{b}_1) > 0 \cup side(\vec{b}_1, \vec{b}_5) > 0$ $\cup\, side(\vec{b}_3, \vec{b}_7) < 0 \cup side(\vec{b}_8, \vec{b}_5) < 0 \cup side(\vec{b}_3, \vec{b}_2) > 0$
(P, N, P)	$(\vec{s}_1)_x > (\vec{b}_{upper})_x \cup (\vec{s}_1)_y < (\vec{b}_{lower})_y \cup (\vec{s}_1)_z > (\vec{b}_{upper})_z$ $\cup\, side(\vec{b}_7, \vec{b}_8) < 0 \cup side(\vec{b}_2, \vec{b}_1) > 0 \cup side(\vec{b}_4, \vec{b}_8) > 0$ $\cup\, side(\vec{b}_2, \vec{b}_6) < 0 \cup side(\vec{b}_4, \vec{b}_1) < 0 \cup side(\vec{b}_7, \vec{b}_6) > 0$
(P, P, N)	$(\vec{s}_1)_x > (\vec{b}_{upper})_x \cup (\vec{s}_1)_y > (\vec{b}_{upper})_y \cup (\vec{s}_1)_z < (\vec{b}_{lower})_z$ $\cup\, side(\vec{b}_6, \vec{b}_5) < 0 \cup side(\vec{b}_3, \vec{b}_4) > 0 \cup side(\vec{b}_1, \vec{b}_5) > 0$ $\cup\, side(\vec{b}_3, \vec{b}_7) < 0 \cup side(\vec{b}_7, \vec{b}_6) < 0 \cup side(\vec{b}_4, \vec{b}_1) > 0$
(P, P, P)	$(\vec{s}_1)_x > (\vec{b}_{upper})_x \cup (\vec{s}_1)_y > (\vec{b}_{upper})_y \cup (\vec{s}_1)_z > (\vec{b}_{upper})_z$ $\cup\, side(\vec{b}_6, \vec{b}_5) < 0 \cup side(\vec{b}_3, \vec{b}_4) > 0 \cup side(\vec{b}_4, \vec{b}_8) > 0$ $\cup\, side(\vec{b}_2, \vec{b}_6) < 0 \cup side(\vec{b}_3, \vec{b}_2) < 0 \cup side(\vec{b}_8, \vec{b}_5) > 0$

that is, $\vec{k}_p$ and $\vec{k}_q$ are nonzero vectors perpendicular to $(\vec{p}_2 - \vec{p}_1)$ and $(\vec{q}_2 - \vec{q}_1)$, respectively. If we apply a dot product by $\vec{k}_p$ on both sides of Eq. (2.26), then the term multiplying t_p evaluates to zero, and we can therefore determine m_q as

$$m_q = \frac{\vec{k}_p \cdot (\vec{p}_1 - \vec{q}_1)}{\vec{k}_p \cdot (\vec{q}_2 - \vec{q}_1)}.$$

If $m_q < 0$ or $m_q > 1$, then the intersection point lies outside the line segment s_2, and the segments do not intersect. Otherwise, we apply a dot product by $\vec{k}_q$ on both sides of Eq. (2.26), and obtain t_p as

$$t_p = \frac{\vec{k}_p \cdot (\vec{q}_1 - \vec{p}_1)}{\vec{k}_q \cdot (\vec{p}_2 - \vec{p}_1)}.$$

Table 2.5 Simplified expressions for each $side(ray, edge)$ relation when the box is axis-aligned with the coordinate frame. The ray direction is given by $\vec{d} = (\vec{s}_2 - \vec{s}_1)$ and the auxiliary variables are computed as $\vec{b}_l = (\vec{b}_{lower} - \vec{s}_1)$ and $\vec{b}_u = (\vec{b}_{upper} - \vec{s}_1)$

side relation	Simplified test
$side(ray, (\vec{b}_3, \vec{b}_4))$	$d_x(\vec{b}_l)_y - d_y(\vec{b}_u)_x$
$side(ray, (\vec{b}_6, \vec{b}_5))$	$d_x(\vec{b}_u)_y - d_y(\vec{b}_l)_x$
$side(ray, (\vec{b}_2, \vec{b}_6))$	$d_x(\vec{b}_u)_z - d_z(\vec{b}_l)_x$
$side(ray, (\vec{b}_4, \vec{b}_8))$	$d_x(\vec{b}_l)_z - d_z(\vec{b}_u)_x$
$side(ray, (\vec{b}_8, \vec{b}_5))$	$d_y(\vec{b}_l)_z - d_z(\vec{b}_u)_y$
$side(ray, (\vec{b}_3, \vec{b}_2))$	$d_y(\vec{b}_u)_z - d_z(\vec{b}_l)_y$
$side(ray, (\vec{b}_3, \vec{b}_7))$	$d_x(\vec{b}_u)_z - d_z(\vec{b}_u)_x$
$side(ray, (\vec{b}_1, \vec{b}_5))$	$d_x(\vec{b}_l)_z - d_z(\vec{b}_l)_x$
$side(ray, (\vec{b}_4, \vec{b}_1))$	$d_y(\vec{b}_l)_z - d_z(\vec{b}_l)_y$
$side(ray, (\vec{b}_7, \vec{b}_6))$	$d_y(\vec{b}_u)_z - d_z(\vec{b}_u)_y$
$side(ray, (\vec{b}_2, \vec{b}_1))$	$d_x(\vec{b}_l)_y - d_y(\vec{b}_l)_x$
$side(ray, (\vec{b}_7, \vec{b}_8))$	$d_x(\vec{b}_u)_y - d_y(\vec{b}_u)_x$

Again, if $t_p < 0$ or $t_p > 1$, then the intersection lies outside the line segment s_1, and the segments do not intersect. Otherwise, the segments intersect at the intersection point computed by substituting either t_p or m_q into Eq. (2.26).

2.5.13 Point-in-Object Test

The determination of whether a point lies inside an object[5] can be efficiently done with the use of six auxiliary rays with origin at the point and directions parallel to the positive and negative axis of the world-coordinate frame, respectively. The point is inside the object if and only if all six rays intersect the object from the inside, as shown in Fig. 2.40.

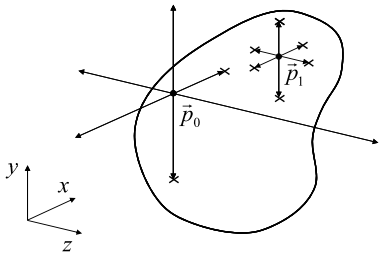

Fig. 2.40 Six auxiliary rays are used to detect whether a point is inside or outside the object. In this example, all rays shot from point $\vec{p}_1$ intersect the object's geometry from the inside, and the point is determined to be inside the object. If any of the rays miss the object or intersect it from the outside, like some shot from point $\vec{p}_0$, then the point is determined to be outside the object

[5] In this book we assume objects are represented by closed meshes.

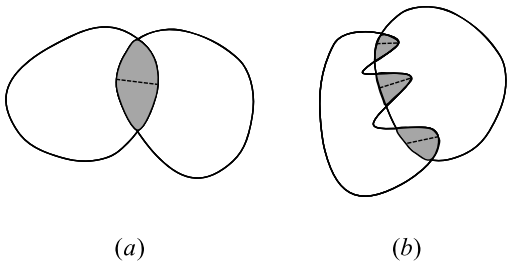

Fig. 2.41 (**a**) Convex objects overlap in only one region; (**b**) Non-convex objects can overlap in multiple disjoint regions. The penetration depth of each region is approximated by the distance between their deepest points

The ray–object intersection tests can be efficiently implemented using the object's hierarchical representation. In a top-down recursion, we start intersecting the ray with the root bounding volume of the hierarchy. If the ray misses the root bounding volume then the point is guaranteed to be outside the object. Otherwise, we recursively check for intersections between the ray and its children node, until the ray misses all children nodes or hits a leaf node. When a leaf node is reached, we perform the appropriate ray–primitive intersection test to determine the intersection point. We also need to check if the ray is hitting the primitive from the inside, that is, the ray is coming out of the object from the intersection point. This can be quickly done by making sure the scalar product between the ray direction and the primitive normal is positive, before computing the intersection point.

2.5.14 Vertex-in-Object Test

Whenever two non-convex objects intersect, we need to determine which vertices of the first object are inside the second object, and vice-versa. This information is used to compute the penetration depth for each disjoint intersection region, as exemplified in Fig. 2.41.

The penetration depth of an intersection region is approximated by the distance between its deepest vertices. In order to determine the deepest vertex of one object inside the other object, we first need to compute all inside vertices associated with the intersection region and then choose the deepest one from this set. This can be done as follows.

Each intersection region has a corresponding intersection curve. The intersection curve is made up of line segments computed from the primitive–primitive intersections. Therefore, each intersection curve can be associated with a set of primitive pairs that intersect at some of the curve's line segments. Moreover, each intersection curve partitions one object into two disjoint regions with respect to the other object. The first region is inside the other object and the second region is outside. So, the vertices of the primitive pairs associated with the intersection curve can be partitioned into two groups as well, namely, the inside and outside groups with respect to the other object. This partition is obtained by executing the point-in-object test described in Sect. 2.5.13, for each vertex of the intersecting primitive pairs to determine the ones that are inside the other object. A simple 2D example of this partition is shown in Fig. 2.42.

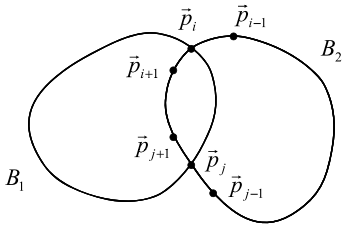

Fig. 2.42 Simple 2D example demonstrating how the intersection information can be used to quickly identify all inside vertices. Objects B_1 and B_2 are intersecting at points $\vec{p}_i$ and $\vec{p}_j$, and we need to determine the vertices of B_2 that are inside B_1. Using the edge connectivity information from B_2's mesh, we can run point-in-object tests for all $\vec{p}_i$'s neighbor vertices. In this example, vertex $\vec{p}_{i-1}$ is determined to be outside B_1, whereas vertex $\vec{p}_{i+1}$ is inside. We recursively inside-test all neighbor vertices of $\vec{p}_{i+1}$ that have not been tested yet until all inside vertices are found

Starting with the list L_1 of vertices of the first object that are inside the second object, we continue searching for neighbor vertices that are also inside the second object until all inside vertices have been found. The easiest way to perform this search is to maintain an auxiliary list L_a of neighbor vertices that were found but still need to be inside-tested against the second object. This auxiliary list is initialized by looping through L_1's vertices and adding all of their neighbor vertices that haven't been tested yet. Then, for each vertex in L_a, we use the point-in-object test to check if the vertex is inside the second object. The vertex is discarded if it lies outside the object. Otherwise, the vertex is added to L_1 and its neighbor vertices that haven't been tested yet are added to L_a. We repeat this process until the auxiliary list L_a is empty, at which point we have determined in L_1 all vertices of the first object that are inside the second object. We go through the same steps to build the list L_2 with all vertices of the second object that are inside the first one. The penetration depth is computed as the distance between the deepest vertices in L_1 and L_2.

2.5.15 Computing Continuous Triangle–Triangle Intersections

The continuous triangle–triangle intersection test takes into account the motion of the triangles for the entire time interval $[t_0, t_1]$. This requires that the actual nonlinear motion of the triangles obtained from the numerical integration module be linearized by assuming the triangles move with constant velocity for the time interval. That is, the trajectories of each triangle vertex are approximated by straight-line segments connecting their positions at t_0 and t_1. Even though this simplification is quite effective in practice, there is a theoretical drawback to it. As illustrated in Fig. 2.43, the in-between motion of the triangles is continuous but not necessary rigid, depending on the amount of rotation experienced by the triangles. Hence, it is possible that the linearized trajectory does not completely bound the original trajectory and thus collisions can still be missed.

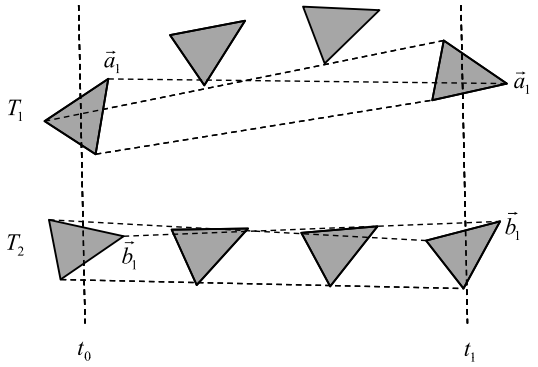

Fig. 2.43 The non-linear motion of triangles T_1 and T_2 obtained from their numerical integration is linearized for continuous collision detection. The vertices are assumed to have moved along a straight line connecting their initial and final positions at t_0 and t_1, respectively. Notice the quality of the linearization degrades as the amount of rotation experienced by the triangles increases. For instance, the linear motion for T_2 better approximates its real motion when compared to the results obtained for triangle T_1, which has a significant rotation component to its motion

Let T_1 and T_2 be the triangles being checked for continuous intersection. For simplicity, let the time parameter be normalized from $[t_0, t_1]$ to $[0, 1]$, with 0 corresponding to the positions at t_0 and 1 at t_1. The triangles will collide during their motion if there is a time $t_c \in [0, 1]$ at which one of the following two cases happen:

1. A vertex of one triangle moves from the front side to the back side of the plane defined by the other triangle at t_c, and the intersection point on the plane lies inside the triangle. This is commonly referred to as the *vertex–face* intersection test.
2. An edge of one triangle crosses an edge of the other triangle at t_c, and their intersection point is inside the end points of both edges. This is known as the *edge–edge* intersection test.

In the vertex–face case, we need to test each vertex of T_1 against T_2, and vice-versa. Since each triangle has three vertices, we have a total of six vertex–face intersection tests to be made. As for the edge–edge case, we need to test an edge of T_1 against all other edges of T_2, and vice-versa. Since each triangle has three edges, we have a total of nine edge–edge tests to be made. Therefore, the continuous triangle–triangle intersection requires a total of fifteen tests, each of them comprising four vertices at a time. These tests are carried out in different ways depending on whether:

1. The triangles' motion is not coplanar.
2. The triangles' motion is coplanar, but the motion of the vertices being tested is not collinear.
3. The triangles' motion is coplanar and the motion of the vertices being tested is collinear.

In each of the above cases, the positions of the four vertices being considered are described by a linear function of time. The geometric relationship needed for the vertices to collide can be rewritten as a polynomial in t. The problem of deter-

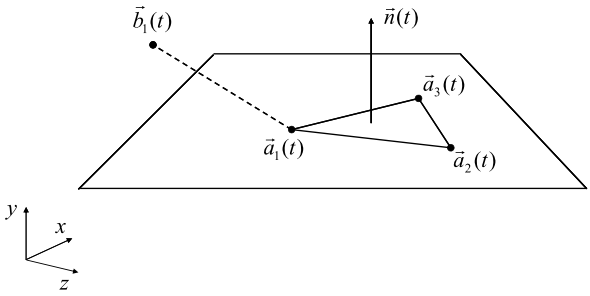

Fig. 2.44 The vertex–face intersection test consists of checking if there is a time t in which the triangle $(\vec{a}_1(t), \vec{a}_2(t), \vec{a}_3(t))$ becomes coplanar with vertex $\vec{b}_1(t)$. The triangle normal is used as the plane normal

mining the collision time is then transformed into a root finding problem for such polynomial. The collision time t_c is set as the smallest real root between zero and one for all fifteen tests. If no real roots between zero and one are found, then the triangles do not intersect.

Case 1: Motion Not Coplanar A necessary intersection condition when the motion of the triangles is not coplanar is that the four vertices being tested for intersections become coplanar at t_c. Otherwise, there can't be an intersection between them in the first place. Hence, their coplanarity is a key geometric condition to determine the collision time t_c in this case.

Let triangles T_1 and T_2 be defined by their vertices $\vec{a}_i(t)$ and $\vec{b}_i(t)$, with $i \in 1, 2, 3$ and $t \in [0, 1]$, such that

$$\begin{aligned} \vec{a}_i(t) &= \vec{a}_i(0) + t\big(\vec{a}_i(1) - \vec{a}_i(0)\big) \\ \vec{b}_i(t) &= \vec{b}_i(0) + t\big(\vec{b}_i(1) - \vec{b}_i(0)\big). \end{aligned} \tag{2.28}$$

Lets first consider the vertex–face intersection test shown in Fig. 2.44. In this case, the face normal is used as the normal of the plane containing all four vertices, that is

$$\vec{n}(t) = \big(\vec{a}_2(t) - \vec{a}_1(t)\big) \times \big(\vec{a}_3(t) - \vec{a}_2(t)\big).$$

Consider the vector connecting vertices $\vec{a}_1(t)$ and $\vec{b}_1(t)$, and notice that it must be perpendicular to the plane normal for the four vertices to be coplanar, that is

$$\vec{n}(t) \cdot \big(\vec{b}_1(t) - \vec{a}_1(t)\big) = 0. \tag{2.29}$$

Expanding the terms in Eq. (2.29) as a function of the vertex positions given in Eq. (2.28), we obtain a 3rd degree polynomial on t of the form:

$$f_3 t^3 + f_2 t^2 + f_1 t + f_0 = 0, \tag{2.30}$$

with the polynomial coefficients f_i given by:

$$\begin{aligned} f_3 &= \vec{k}_1 \cdot \vec{k}_2 \\ f_2 &= \vec{k}_3 \cdot \vec{k}_2 + \vec{k}_1 \cdot \vec{k}_4 \\ f_1 &= \vec{k}_3 \cdot \vec{k}_4 + \vec{k}_1 \cdot \vec{k}_5 \\ f_0 &= \vec{k}_3 \cdot \vec{k}_5, \end{aligned} \tag{2.31}$$

and the auxiliary variables $\vec{k}_i$ obtained from the vertex positions at 0 and 1, namely:

$$\begin{aligned} \vec{k}_1 &= \big(\vec{b}_1(1) - \vec{b}_1(0)\big) - \big(\vec{a}_1(1) - \vec{a}_1(0)\big) \\ \vec{k}_2 &= \Big(\big(\vec{a}_2(1) - \vec{a}_2(0)\big) - \big(\vec{a}_1(1) - \vec{a}_1(0)\big)\Big) \\ &\quad \times \Big(\big(\vec{a}_3(1) - \vec{a}_3(0)\big) - \big(\vec{a}_2(1) - \vec{a}_2(0)\big)\Big) \\ \vec{k}_3 &= \vec{b}_1(0) - \vec{a}_1(0) \\ \vec{k}_4 &= \big(\vec{a}_2(0) - \vec{a}_1(0)\big) \times \Big(\big(\vec{a}_3(1) - \vec{a}_3(0)\big) - \big(\vec{a}_2(1) - \vec{a}_2(0)\big)\Big) \\ &\quad + \Big(\big(\vec{a}_2(1) - \vec{a}_2(0)\big) - \big(\vec{a}_1(1) - \vec{a}_1(0)\big)\Big) \times \big(\vec{a}_3(0) - \vec{a}_2(0)\big) \\ \vec{k}_5 &= \big(\vec{a}_2(0) - \vec{a}_1(0)\big) \times \big(\vec{a}_3(0) - \vec{a}_2(0)\big). \end{aligned}$$

There are a few different ways for computing the roots of the polynomial in Eq. (2.30), varying from closed formulas to advanced iterative methods. We refer the reader to Sect. 2.6 for references to the literature where these different solution methods can be found. Here, we assume the roots were computed using one of such methods.

For each real root t_r found in the $[0, 1]$ interval, we need to verify that an actual intersection does take place. This is done by positioning the four vertices at t_r and performing a point-in-triangle test to make sure vertex $\vec{b}_1(t_r)$ is inside triangle $(\vec{a}_1(t_r), \vec{a}_2(t_r), \vec{a}_3(t_r))$. If the vertex lies outside the triangle, then the points are coplanar but do not intersect at t_r and this root is discarded. Otherwise, the current collision time t_c is updated to t_r if $t_r \leq t_c$. The collision point and normal are set as $\vec{b}_1(t_r)$ and $\vec{n}(t_r)$, respectively.

Now, lets consider the edge–edge intersection test shown in Fig. 2.45. The main difference between this case and the vertex–face case is in the way the plane normal is computed. Since we no longer have an actual face to use its normal vector as the plane normal, and the plane normal needs to be perpendicular to both edges, we use the cross-product of the edges as the plane normal instead, that is

$$\vec{n}(t) = \big(\vec{a}_2(t) - \vec{a}_1(t)\big) \times \big(\vec{b}_2(t) - \vec{b}_1(t)\big).$$

Again, the vector connecting vertices $\vec{a}_1(t)$ and $\vec{b}_1(t)$ must be perpendicular to the plane normal in order to have all four vertices coplanar, that is

$$\vec{n}(t) \cdot \big(\vec{b}_1(t) - \vec{a}_1(t)\big) = 0. \tag{2.32}$$

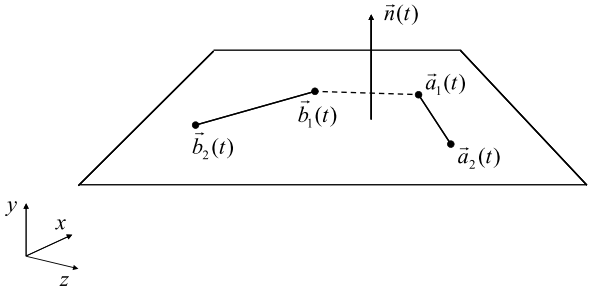

Fig. 2.45 In the edge–edge intersection test, the edges $(\vec{a}_1(t), \vec{a}_2(t))$ and $(\vec{b}_1(t), \vec{b}_2(t))$ are checked for coplanarity as well. The plane normal must be perpendicular to both edges and is obtained from their cross-product

The polynomial coefficients f_i in the edge–edge case remain the same as in Eq. (2.31), but the auxiliary variables $\vec{k}_i$ are updated to:

$$\begin{aligned}
\vec{k}_1 &= \big(\vec{b}_1(1) - \vec{b}_1(0)\big) - \big(\vec{a}_1(1) - \vec{a}_1(0)\big) \\
\vec{k}_2 &= \big(\big(\vec{a}_2(1) - \vec{a}_2(0)\big) - \big(\vec{a}_1(1) - \vec{a}_1(0)\big)\big) \times \big(\big(\vec{b}_2(1) - \vec{b}_2(0)\big) - \big(\vec{b}_1(1) - \vec{b}_1(0)\big)\big) \\
\vec{k}_3 &= \vec{b}_1(0) - \vec{a}_1(0) \\
\vec{k}_4 &= \big(\vec{a}_2(0) - \vec{a}_1(0)\big) \times \big(\big(\vec{b}_2(1) - \vec{b}_2(0)\big) - \big(\vec{b}_1(1) - \vec{b}_1(0)\big)\big) \\
&\quad + \big(\big(\vec{a}_2(1) - \vec{a}_2(0)\big) - \big(\vec{a}_1(1) - \vec{a}_1(0)\big)\big) \times \big(\vec{b}_2(0) - \vec{b}_{21}(0)\big) \\
\vec{k}_5 &= \big(\vec{a}_2(0) - \vec{a}_1(0)\big) \times \big(\vec{b}_2(0) - \vec{b}_1(0)\big).
\end{aligned}$$

Again, for each real root t_r found in the $[0, 1]$ interval, we need to position the two edges at that time and do an edge–edge intersection test to make sure the edges $(\vec{a}_1(t_r), \vec{a}_2(t_r))$ and $(\vec{b}_1(t_r), \vec{b}_2(t_r))$ do intersect. If the edges are parallel or intersect outside their end points, then they are coplanar but do not intersect and the root t_r is discarded. Otherwise, the current collision time t_c is updated to t_r if $t_r < t_c$. The collision point and normal are set as the intersection point between the edges at t_r and $\vec{n}(t_r)$, respectively.

As far as robustness is concerned, the collision normal obtained in the vertex–face intersection test tends to be more stable than the one obtained in the edge–edge intersection test. In the first case, the collision normal is the face normal itself whereas in the latter case the collision normal comes from the cross-product of the edges. If the edges are parallel or almost aligned, then the cross-product will be zero or very close to it and the computed normal vector becomes less reliable. Therefore, the collision information for the vertex–face case always takes precedence over the one for the edge–edge case when both collision times are the same. That's why we used the condition $t_r \leq t_c$ to update the collision time for the vertex–face case, and $t_r < t_c$ to update it for the edge–edge case.

Case 2: Motion Coplanar but Not Collinear In the special case in which the triangles' motion is coplanar for the entire time interval, the polynomial Eq. (2.30) degenerates because all of its coefficients f_i evaluate to zero. However, the triangles can still intersect while moving on the plane. This coplanar case is illustrated in Fig. 2.46.

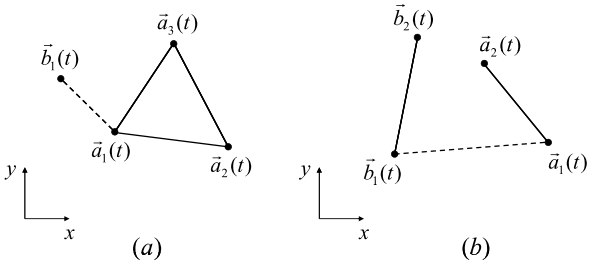

Fig. 2.46 (**a**) The vertex–face coplanar test consists of checking whether there is a time t in which vertex $\vec{b}_1(t)$ and one of the triangle edges become collinear; (**b**) In the edge–edge case, we need to test if the vertices of one edge become collinear with the other edge, and vice-versa. Notice that in both cases, the plane normal is aligned with the positive z-axis (pointing towards the reader)

Each coplanar vertex–face intersection test is replaced by three vertex–edge intersection tests in which a vertex of one triangle and an edge of the other triangle are tested for collinearity. Since we have six vertex–face tests to be made, we end up with a total of eighteen vertex–edge collinearity tests instead. Similarly, each coplanar edge–edge intersection test is replaced by four vertex–edge tests, in which a vertex of one edge is checked for collinearity with the other edge. Fortunately, all vertex–edge collinearity tests for the edge–edge case are a subset of the ones used in the vertex–face case, so they don't add up to the total of eighteen tests needed.

The collinearity tests are best carried out in a local-coordinate frame with the plane normal aligned with one of the coordinate axis, for example, the positive local z-axis. Consider a world-to-local transformation matrix with origin at $\vec{b}_1(0)$ and a rotation component that makes the plane normal in world-frame be the positive z-axis in local-frame. Let the vertices $\vec{a}_i(t)$ and $\vec{b}_i(t)$ in world-frame become vertices $\vec{la}_i(t)$ and $\vec{lb}_i(t)$ in local-frame under this world-to-local transformation. Since both vertex–face and edge–edge tests are replaced by a combination of vertex–edge tests, let's assume vertex $\vec{lb}_1(t)$ is being tested for collinearity with edge $(\vec{la}_1(t), \vec{la}_2(t))$. The vertices will be collinear whenever

$$\left(\vec{la}_2(t) - \vec{la}_1(t)\right) \times \left(\vec{lb}_1(t) - \vec{la}_2(t)\right) = 0. \tag{2.33}$$

Expanding the terms in Eq. (2.33) as a function of the transformed vertex positions, we obtain the following 2nd degree polynomial on t

$$f_2 t^2 + f_1 t + f_0 = 0, \tag{2.34}$$

with the polynomial coefficients f_i given by:

$$\begin{aligned} f_2 &= \vec{k}_1 \times \vec{k}_2 \\ f_1 &= \vec{k}_3 \times \vec{k}_2 + \vec{k}_1 \times \vec{k}_4 \\ f_0 &= \vec{k}_3 \times \vec{k}_4, \end{aligned} \tag{2.35}$$

and the auxiliary variables $\vec{k}_i$ obtained from the transformed vertex positions at 0 and 1, namely:

$$\begin{aligned}
\vec{k}_1 &= \left(\vec{la}_2(1) - \vec{la}_2(0)\right) - \left(\vec{la}_1(1) - \vec{la}_1(0)\right) \\
\vec{k}_2 &= \vec{lb}_1(1) - \left(\vec{la}_2(1) - \vec{la}_2(0)\right) \\
\vec{k}_3 &= \vec{la}_2(0) - \vec{la}_1(0) \\
\vec{k}_4 &= -\vec{la}_2(0).
\end{aligned}$$

The roots of the polynomial in Eq. (2.34) can be found analytically. For each real root $t_r \in [0, 1]$, we need to position both vertex and edge at that time and do a point-in-edge intersection test to make sure the vertex is inside the end points of the edge. If the vertex is outside, then the vertex and the edge are collinear but do not intersect and the root t_r is discarded. Otherwise, the current collision time t_c is updated to t_r if $t_r \leq t_c$, and the collision point and normal is obtained from the current vertex–edge position.

Case 3: Motion Coplanar and Collinear In the rare but still feasible situation in which the vertex and the edge in the above case are collinear for their entire motion, all coefficients of the polynomial Eq. (2.34) evaluate to zero. In this case, vertex $\vec{lb}_1(t)$ will intersect edge $(\vec{la}_1(t), \vec{la}_2(t))$ only if it passes through one of the edge's end points during its motion. This condition equates to the following two 1st degree polynomial equations on t

$$\begin{aligned}
\vec{lb}_1(t) &= \vec{la}_1(t) \\
\vec{lb}_1(t) &= \vec{la}_2(t).
\end{aligned} \tag{2.36}$$

Expanding the terms in Eq. (2.36) as a function of the transformed vertex positions, we obtain the following two possible intersection times

$$t_r = \frac{(\vec{lb}_1(0) - \vec{la}_i(0))}{(\vec{la}_i(1) - \vec{la}_i(0)) - (\vec{lb}_1(1) - \vec{lb}_1(0))}$$

with $i \in 1, 2$. Any of these roots are discarded if they are below zero or above one. The current collision time t_c is updated to the smallest remaining root t_r if $t_r \leq t_c$, and the collision point and normal is obtained from the current vertex–edge position.

2.5.16 Computing Continuous Sphere–Sphere Intersections

The continuous sphere–sphere intersection test consists of determining the earliest time $t \in [0, 1]$ at which the distance between the center of the spheres becomes less than the sum of their radiuses. Let the spheres be defined by their center points $\vec{a}(t)$

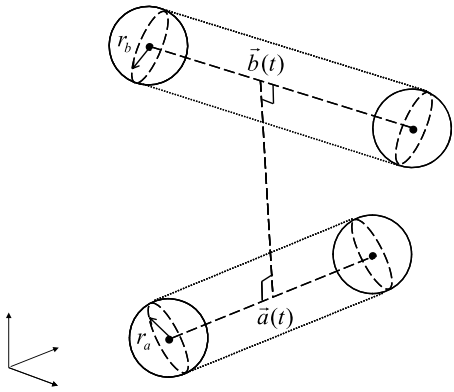

Fig. 2.47 Two spheres being tested for continuous collision. We need to find the earliest time t at which their centers are distant by less than the sum of their radiuses

and $\vec{b}(t)$ and radiuses r_a and r_b, as shown in Fig. 2.47. The position of the centers at time t is given by

$$\begin{aligned}\vec{a}(t) &= \vec{a}(0) - t\big(\vec{a}(1) - \vec{a}(0)\big)\\ \vec{b}(t) &= \vec{b}(0) - t\big(\vec{b}(1) - \vec{b}(0)\big).\end{aligned} \tag{2.37}$$

We begin by testing if the spheres are already intersecting at $t = 0$, that is, checking if

$$\big(\vec{a}(0) - \vec{b}(0)\big) \cdot \big(\vec{a}(0) - \vec{b}(0)\big) \leq (r_a + r_b)^2. \tag{2.38}$$

If Eq. (2.38) is satisfied, then the intersection time is set to t_0 and the collision normal is taken as the line connecting the spheres' center points. Otherwise, we need to find the smallest $t \in [0, 1]$, if any, that satisfies the condition

$$\big(\vec{a}(t) - \vec{b}(t)\big) \cdot \big(\vec{a}(t) - \vec{b}(t)\big) \leq (r_a + r_b)^2. \tag{2.39}$$

Substituting Eq. (2.37) into Eq. (2.39) and regrouping terms, we obtain the following 2nd degree polynomial on t

$$f_2 t^2 + f_1 t + f_0 = 0, \tag{2.40}$$

with the polynomial coefficients f_i given by:

$$\begin{aligned}f_2 &= \big(\big(\vec{a}(1) - \vec{a}(0)\big) - \big(\vec{b}(1) - \vec{b}(0)\big)\big)\\ &\quad \times \big(\big(\vec{a}(1) - \vec{a}(0)\big) - \big(\vec{b}(1) - \vec{b}(0)\big)\big)\\ f_1 &= 2\big(\vec{a}(0) - \vec{b}(0)\big) \cdot \big(\big(\vec{a}(1) - \vec{a}(0)\big) - \big(\vec{b}(1) - \vec{b}(0)\big)\big)\\ f_0 &= \big(\vec{a}(0) - \vec{b}(0)\big) \cdot \big(\vec{a}(0) - \vec{b}(0)\big) - (r_a + r_b)^2.\end{aligned} \tag{2.41}$$

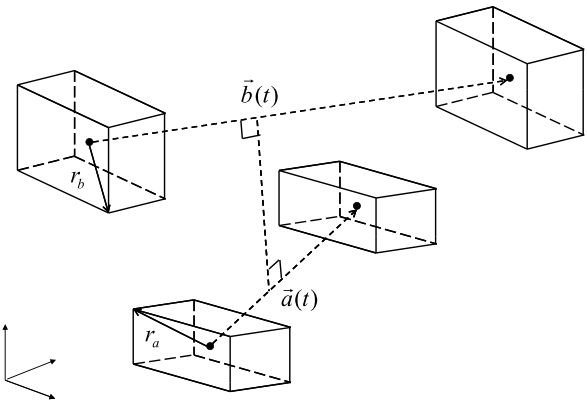

Fig. 2.48 Two boxes moving from t_0 to t_1. Their continuous intersection test can be split into three one-dimensional intersection tests along each coordinate axis

The roots of the polynomial in Eq. (2.40) can be found analytically. The spheres will intersect only if there are roots between 0 and 1. The smallest of such roots is set as the collision time for the spheres, and the collision normal is set as the line connecting the current position of their center points.

2.5.17 Computing Continuous Box–Box Intersections

The continuous box–box intersection test is very similar to the continuous sphere–sphere intersection test described in the previous section. The boxes will intersect at the earliest time $t \in [0, 1]$ in which the distance between their centers is less than the sum of their half-radiuses along each axis. Since this condition can be enforced independently along each axis, the continuous box–box intersection test can be split into three one-dimensional tests.

Let the boxes be defined by their center points $\vec{a}(t)$ and $\vec{b}(t)$ and half radiuses $\vec{r}_a$ and $\vec{r}_b$, respectively, as shown in Fig. 2.48. The position of the centers at time t is given by

$$\begin{aligned} \vec{a}(t) &= \vec{a}(0) - t\big(\vec{a}(1) - \vec{a}(0)\big) \\ \vec{b}(t) &= \vec{b}(0) - t\big(\vec{b}(1) - \vec{b}(0)\big). \end{aligned} \tag{2.42}$$

We begin by testing whether the boxes are already intersecting at $t = 0$, that is, if the following three conditions are simultaneously satisfied:

$$\begin{aligned} \big(\vec{a}(0) - \vec{b}(0)\big)_x &\leq (\vec{r}_a + \vec{r}_b)_x \\ \big(\vec{a}(0) - \vec{b}(0)\big)_y &\leq (\vec{r}_a + \vec{r}_b)_y \\ \big(\vec{a}(0) - \vec{b}(0)\big)_z &\leq (\vec{r}_a + \vec{r}_b)_z. \end{aligned}$$

If this is the case, then the intersection time is set to t_0, and the collision normal is computed from the line connecting the boxes' center points. Otherwise, we need to find the smallest $t \in [0, 1]$, if any, that concurrently satisfies the conditions:

$$\begin{aligned}\big(\vec{a}(t) - \vec{b}(t)\big)_x &\leq (\vec{r}_a + \vec{r}_b)_x \\ \big(\vec{a}(t) - \vec{b}(t)\big)_y &\leq (\vec{r}_a + \vec{r}_b)_y \\ \big(\vec{a}(t) - \vec{b}(t)\big)_z &\leq (\vec{r}_a + \vec{r}_b)_z.\end{aligned} \tag{2.43}$$

Substituting Eq. (2.42) into Eq. (2.43), we obtain three 2nd degree polynomial equations on t that must be simultaneously satisfied, namely:

$$\begin{aligned}(f_2)_x t^2 + (f_1)_x t + (f_0)_x &= 0 \\ (f_2)_y t^2 + (f_1)_y t + (f_0)_y &= 0 \\ (f_2)_z t^2 + (f_1)_z t + (f_0)_z &= 0,\end{aligned}$$

where the polynomial coefficients are given by:

$$\begin{aligned}(f_2)_i &= \big(\big(\vec{a}(1) - \vec{a}(0)\big)_i - \big(\vec{b}(1) - \vec{b}(0)\big)_i\big)^2 \\ (f_1)_i &= 2\big(\vec{a}(0) - \vec{b}(0)\big)_i \big(\big(\vec{a}(1) - \vec{a}(0)\big)_i - \big(\vec{b}(1) - \vec{b}(0)\big)_i\big) \\ (f_0)_i &= \big(\vec{a}(0) - \vec{b}(0)\big)_i^2 - (r_a + r_b)_i^2,\end{aligned}$$

with $i \in \{x, y, z\}$ specifying the axis along which the intersection is being checked. For each root between 0 and 1 of each polynomial, we position the boxes at the root time and verify that the objects are in fact intersecting, that is, that their center distance is less than the sum of their half radiuses. The smallest time t_c that satisfy these conditions is set as the intersection time between the boxes.

2.6 Notes and Comments

The literature on hierarchical decompositions is extensive, with related publications on several research areas such as computational geometry, computer graphics, robotics and molecular simulations. There are several other representations and variants of the techniques presented in this chapter, specially with respect to implementation.

The OBB tree representation became an option since Gottschalk et al. [GLM96, Got96] introduced the separating-axis theorem for carrying out fast interference detection between arbitrarily oriented boxes. Bergen [vdB97] presented a modified interference-detection test using AABB tree representations in which the search for a separating axis considered only the normals of the faces of the boxes. The pairwise edge-direction tests were ignored, thus reducing the complexity of the test, but consequently being about 6 % less accurate.

The use of bounding spheres, instead of bounding boxes, is also popular, owing to the simplicity of its implementation. The efficiency of bounding-sphere representations can be further improved if we use quad-trees or oct-trees instead of binary trees. Samet [Sam89] gives a good introduction to both quad-tree and oct-tree representations. The difficulty in using bounding-sphere representations is to come up with a partition that best approximates the original polyhedra. Hubbard [Hub96] developed a collision-detection algorithm that approximates 3D polyhedra with an oct-tree representation of bounding spheres, using a sophisticated technique based on 3D Voronoi diagrams to construct the spheres at each intermediate level of the decomposition.

The 3D convex-hull computation can be found in Preparata et al. [PS85], Edelsbrunner [Ede87], and in several other books on computational geometry. In the OBB case, there is also the need to determine the eigenvectors of the covariance matrix of the vertices of the convex hull. Eigenvectors and their associated eigenvalues are covered in detail in Strang [Str91], Golub [GL96], Horn [HJ91] and Press et al. [PTVF96].

The multi-level grid-structure analysis presented in Sect. 2.4.2 was derived from Mirtich [Mir96b]. Some primitive–primitive tests presented in Sect. 2.5 were obtained from Gottschalk [Got96] (box–box), Arvo [Arv90] and Larsson et al. [LML07] (box–sphere), Ritter [Rit90] (sphere–sphere), Karabassi et al. [KPTB99] (sphere–triangle) and Mahovsky et al. [MW04] (ray–box). The triangle–triangle intersection test presented in Sect. 2.5.5 is a combination of the different intersection tests found in Held [Hel97], Möller [Möl97] and Glaeser [Gla94]. Other interesting primitive–primitive tests can be found in Held [Hel97] and Ericson [Eri05].

The continuous time triangle–triangle intersection was originally introduced by Provot [Pro97] in the context of cloth simulations. Redon et al. [RKC02, RL06] used the concept of a screw motion to linearize the translation and rotation for the time interval being checked for collisions. They also present an interval-arithmetic approach to compute the roots of a 3rd degree polynomial which takes rounding errors into account. A closed form calculation of the roots can be found in Schwarze [Sch93].

2.7 Exercises

1. How can we improve the culling efficiency in the hierarchy–hierarchy intersection tests if we have information about the volume of its internal nodes?
2. Derive an algorithm to parallelize the hierarchy-self intersection algorithm presented in Sect. 2.5.2. What is the expected execution time of the algorithm?
3. Consider a set of points in world-space and a world-to-local transformation with origin at their mean point and local-coordinate axis aligned with the eigenvectors of the covariance matrix associated with the points. Consider two bounding spheres, one computed in world-space and the other in local-space using the above world-to-local transformation. Which bounding sphere has a tighter fit around the points? Explain.

4. The continuous triangle–triangle intersection test presented in Sect. 2.5.15 can be optimized in many ways.
   (a) Derive an equation for the vertex–face case to perform a rejection test if the vertex motion is on the same side of the plane containing the triangle. Derive an equivalent equation for the edge–edge case.
   (b) In a triangulated mesh, an edge is shared by two faces, whereas a vertex is shared by an average of six faces. Whenever a collision occurs at an edge or vertex, the collision detection will report multiple face–face collision candidates for the same collision. In particular, it will report two face–face collision candidates if the collision occurs at an edge, and as many face–face candidates as there are faces incident on a vertex, if the collision occurs at the vertex. Each of these multiple face–face collision candidates will be tested for intersection, reporting back the same intersection result. Devise a memory efficient data structure that is smart enough to process collisions at a vertex or edge just once, avoiding unnecessary, time-consuming, duplicate work.
5. We can use Sturm's theorem to find the number of real roots in the [0, 1] interval for the polynomials obtained in the continuous collisions described in Sects. 2.5.15, 2.5.16 and 2.5.17. The idea is to perform a quick rejection test and avoid solving the polynomial equations if there are no real roots in this interval.
   (a) Derive the equations for the Sturm chain for quadratic and cubic polynomials.
   (b) Is the quick rejection test still valid if an element of the chain evaluates to zero?
6. In the context of computing polynomial roots, consider the special case in which we have a double real root in the [0, 1] interval. Assume that the root was *not* found due to numerical rounding errors in the calculations. How can we improve the robustness of the root finding process to not miss such intersection cases? (*Hint*: saddle points.)

References

[Arv90] Arvo, J.: A simple method for box-sphere intersection testing. In: Graphics Gems I, pp. 335–339 (1990)

[Ede87] Edelsbrunner, H.: Algorithms in Combinatorial Geometry. Springer, Berlin (1987)

[Eri05] Ericson, C.: Real-Time Collision Detection. Kaufmann, Los Altos (2005)

[GL96] Golub, G.H., Van Loan, C.F.: Matrix Computations. Johns Hopkins University Press, Baltimore (1996)

[Gla94] Glaeser, G.: Fast Algorithms for 3D-Graphics. Springer, Berlin (1994)

[GLM96] Gottschalk, S., Lin, M.C., Manocha, D.: Obbtree: a hierarchical structure for rapid interference detection. Comput. Graph. (Proc. SIGGRAPH) **30**, 171–180 (1996)

[Got96] Gottschalk, S.: The separating axis test. Technical Report TR-96-24, University of North Carolina, Chapel Hill (1996)

[Hel97] Held, M.: Erit—a collection of efficient and reliable intersection tests. J. Graph. Tools **2**(4), 25–44 (1997)

[HJ91] Horn, R.A., Johnson, C.R.: Matrix Analysis. Cambridge University Press, Cambridge (1991)

[Hub96] Hubbard, P.M.: Approximating polyhedra with spheres for time-critical collision detection. ACM Trans. Graph. **15**(3), 179–210 (1996)
[KPTB99] Karabassi, E.-A., Papaioannou, G., Theoharis, T., Boehm, A.: Intersection test for collision detection in particle systems. J. Graph. Tools **4**(1), 25–37 (1999)
[LML07] Larsson, T., Möller, T., Lengyel, E.: On faster sphere-box overlap testing. J. Graph. Tools **12**(1), 3–8 (2007)
[Mir96b] Mirtich, B.V.: Impulse-based dynamic simulation of rigid body systems. PhD Thesis, University of California, Berkeley (1996)
[Möl97] Möller, T.: A fast triangle–triangle intersection test. J. Graph. Tools **2**(2), 25–30 (1997)
[MW04] Mahovsky, J., Wyvill, B.: Fast ray-axis aligned bounding box overlap tests with Plücker coordinates. J. Graph. Tools **9**(1), 35–46 (2004)
[Pro97] Provot, X.: Collision and self-collision handling in cloth model dedicated to design garments. In: Proceedings Graphics Interface, pp. 177–189 (1997)
[PS85] Preparata, F.P., Shamos, M.I.: Computational Geometry: An Introduction. Springer, Berlin (1985)
[PTVF96] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, Cambridge (1996)
[Rit90] Ritter, J.: An efficient bounding sphere. In: Graphics Gems I, pp. 301–303 (1990)
[RKC02] Redon, S., Kheddar, A., Coquillart, S.: Fast continuous collision detection between rigid bodies. Proc. EUROGRAPHICS **21** (2002)
[RL06] Redon, S., Lin, M.C.: A fast method for local penetration depth computation. J. Graph. Tools **11**(2), 37–50 (2006)
[Sam89] Samet, H.: Spatial Data Structures: Quadtree, Octrees and Other Hierarchical Methods. Addison-Wesley, Reading (1989)
[Sch93] Schwarze, J.: Cubic and quartic roots. In: Graphics Gems I, pp. 404–407 (1993)
[Str91] Strang, G.: Linear Algebra and Its Applications. Academic Press, San Diego (1991)
[vdB97] van den Bergen, G.: Efficient collision detection of complex deformable models using AABB trees. J. Graph. Tools **2**(4), 1–13 (1997)

http://www.springer.com/978-1-4471-4416-8

Guide to Dynamic Simulations of Rigid Bodies and Particle Systems

Coutinho, M.G.

2013, XIV, 402 p., Hardcover
ISBN: 978-1-4471-4416-8