Real-Time Rendering
Fourth Edition

Online chapter: Collision Detection

version 1.2; July 16, 2018

Download latest from http://realtimerendering.com

Tomas Akenine-Möller
Eric Haines
Naty Hoffman
Angelo Pesce
Michał Iwanicki
Sébastien Hillaire

Contents

Chapter 25
Collision Detection

"To knock a thing down, especially if it is cocked at an arrogant angle, is a deep delight to the blood."
—George Santayana

Collision detection (CD) is a fundamental and important ingredient in many computer graphics applications. Areas where CD plays a vital role include virtual manufacturing, CAD/CAM, computer animation, physically based modeling, games, flight and vehicle simulators, robotics, path and motion planning (tolerance verification), assembly, and almost all virtual reality simulations. Due to its huge number of uses, CD has been and still is a subject of extensive research.

Collision detection is part of what is often referred to as *collision handling*, which can be divided into three major parts: *collision detection*, *collision determination*, and *collision response*. The result of collision detection is a boolean saying whether two or more objects collide, while collision determination finds the actual intersections between objects; finally, collision response determines what actions should be taken in response to the collision of two objects.

Consider an old-style clock consisting of springs and cog-wheels. Say this clock is represented as a detailed, three-dimensional model in the computer. Now imagine that the clock's movements are to be simulated using collision detection. The spring is wound and the clock's motion is simulated as collisions are detected and responses are generated. Such a system would require collision detection between possibly thousands of pairs of objects. Another challenge would be to simulate all the ants and all the pine needles in an anthill. A brute-force search of all possible collision pairs would be incredibly inefficient and so is impractical under these circumstances. An example of complex collision detection is shown in Figure 25.1.

To cope with large scenes, it is common to divide a collision detection system into three phases. Broad phase CD works on a per object level and finds pairs of objects whose BVs overlap (Section 25.1). Mid phase CD (Section 25.2) continues the work and detects parts of two objects that may overlap, and finally, narrow phase CD (Section 25.3) works on leaves of primitives or on convex parts of objects. These three phases can be used together in a large CD system.

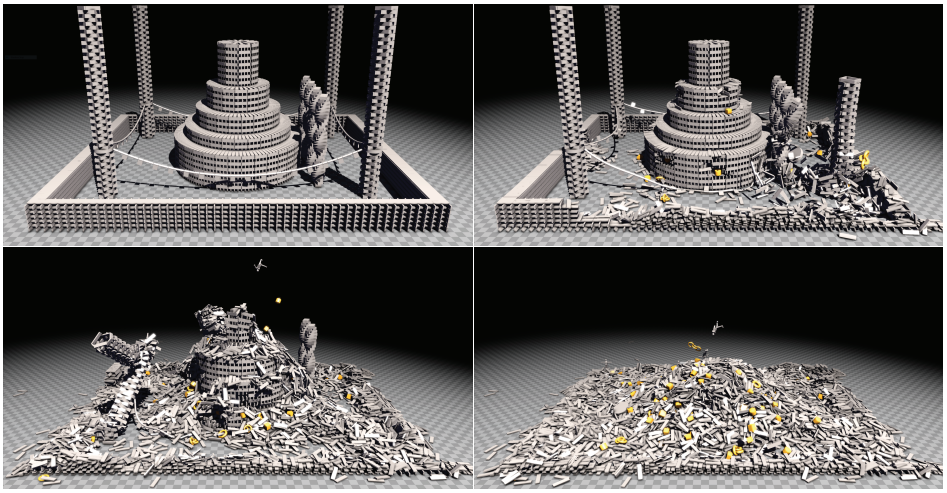

Figure 25.1. Rigid-body collision detection using tens of thousands of small geometrical objects, mostly boxes, torii, and an occasional ragdoll. *(Images generated using the PhysX Kapla demo, courtesy of NVIDIA Corporation.)*

In Section 25.4 we discuss simple and extremely fast collision detection techniques that are useful in some scenarios. The main idea is to approximate a complex object using a set of line segments. These line segments are then tested for intersection with the primitives of the environment. This technique is sometimes used in games. Another approximative method is described in Section 25.5, where a BSP tree representation of the environment is used, and a cylinder may be used to describe a character. However, all objects cannot always be approximated with line segments or cylinders, and some applications may require more accurate tests.

Time-critical collision detection is a technique for doing approximate collision detection in constant time, and is treated in Section 25.6. Then follows sections on deformable models, continuous CD, collision response, and finally, how to handle particles, in Section 25.10.

It must be pointed out that performance evaluations are difficult in the case of CD, since the algorithms are sensitive to the collision scenarios, and there is no algorithm that performs best in all cases [33]. Also, note that CD is often performed on the CPU side, but due to the generality of GPUs, all algorithms can be executed on the GPU as well. In general, algorithms need to be adapted to the particular architecture's memory system and feature set in order to maximize performance.

To conclude, we present some dynamic intersection tests. Determining whether two moving objects collide during a given duration adds literally another dimension to the problem: time. This final section provides common tests for various primitive combinations and tools for deriving more complex interactions.

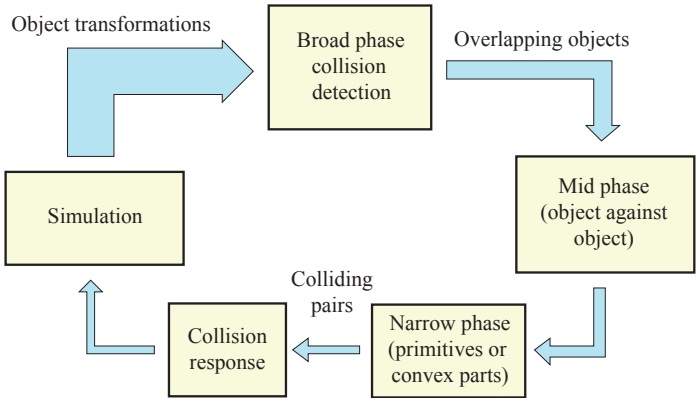

Figure 25.2. A collision detection system that is fed object transformations using some kind of simulation. All objects in a scene are then processed by broad phase CD in order to quickly find objects pairs whose bounding volumes overlap. Next, the mid phase CD continues to work on pairs of objects to find leaves of primitives or convex parts that overlap. Finally, the CD system performs the lowest level operations in the narrow phase, where one can compute primitive-primitive intersections or use distance queries and then feed the result to collision response.

25.1 Broad Phase Collision Detection

In this section, we describe several broad phase CD algorithms. However, first we illustrate the broad, mid, and narrow phase of CD in Figure 25.2, which is a two-level CD system [16] where the last phase has been split up into a mid and a narrow phase. The target is large-scale environments with multiple objects in motion. The broad phase of this system reports potential collisions among all the objects in the environment. Next, the mid phase works on pairs of objects and attempts to find leaves of primitives that potentially overlap or convex parts that overlap, for example. The narrow phase computes primitive-primitive intersections or uses distance queries to make sure that objects do not penetrate each other. Finally, collision response (Section 25.9) is computed, whose result is fed into a simulation subsystem that can compute final transforms for each object.

Since a scene may contain thousands of moving objects, a good CD system must handle such situations well. If the scene contains n moving and m static objects, then a naive method would perform

$$nm + \binom{n}{2} = nm + n(n-1)/2 \tag{25.1}$$

object tests for each frame. The first term corresponds to testing the number of static objects against the dynamic (moving) objects, and the last term corresponds to testing the dynamic objects against each other. The naive approach quickly becomes expensive as m and n rise. This situation calls for smarter methods, which are the subject this section.

The goal of each phase is to minimize the amount of work passed to the subsequent phase, i.e., we want to minimize pairs of objects, primitives, convex parts, etc., that are fed to the next phase. At the first phase, testing is done on the object level and it is therefore often called broad phase CD.

Most algorithms for this phase start by enclosing each object in a BV, and then apply some technique to find all BV/BV pairs that overlap. A simple approach is to use an axis-aligned bounding box (AABB) for each object. To avoid recomputing this AABB for an object undergoing rigid-body motion, the AABB is adjusted to be a *fixed cube* large enough to contain the object in any arbitrary orientation. The fixed cubes are used to rapidly determine which pairs of objects' bounding volumes are disjoint. Sometimes it may be better to use dynamically resized AABBs, which are fast to recompute for oriented boxes, spheres, and capsules, for example.

Spheres can be used instead of fixed cubes. This is reasonable, since the sphere is the perfect BV with which to enclose an object at any orientation. It is also possible to use the apex map [53] (Section 22.13.4). In the following, we describe three algorithms for broad phase CD, namely, sweep-and-prune, using grids, and using BVHs. An entirely different method is to use the loose octree structure presented in Section 19.1.3.

25.1.1 Sweep-and-Prune

We assume that each object has an enclosing AABB. In the *sweep-and-prune* technique [3, 62, 93], *temporal coherence*, which often is present in typical applications, is exploited. Temporal coherence means that objects undergo relatively small (if any) changes in their position and orientation from frame to frame (and so it is also called *frame-to-frame coherence*).

Lin [62] points out that the overlapping bounding box problem in three dimensions can be solved in $O(n \log^2 n + k)$ time (where k is the number of pairwise overlaps), but it can be improved upon by exploiting coherence and so can be reduced to $O(n + k)$. However, this assumes that the animation has a fair amount of temporal coherence.

If two AABBs overlap, all three one-dimensional intervals (formed by the start and end points of AABBs) in each main axis direction must also overlap. Here, we will describe how all overlaps of a number of one-dimensional intervals can be detected efficiently when the frame-to-frame coherency is high. Given that solution, the three-dimensional problem for AABBs is solved by using the one-dimensional algorithm for each of the three main axes.

Assume that n intervals (along a particular axis) are represented by s_i and e_i, where $s_i < e_i$ and $0 \leq i < n$. These values are sorted in one list in increasing order. This list is then swept from start to end. When a start point s_i is encountered, the corresponding interval is put into an active interval list. When an endpoint is encountered, the corresponding interval is removed from the active list. Now, if the start point of an interval is encountered while there are intervals in the active list, then the encountered interval overlaps all intervals in the active list. This is shown in Figure 25.3.

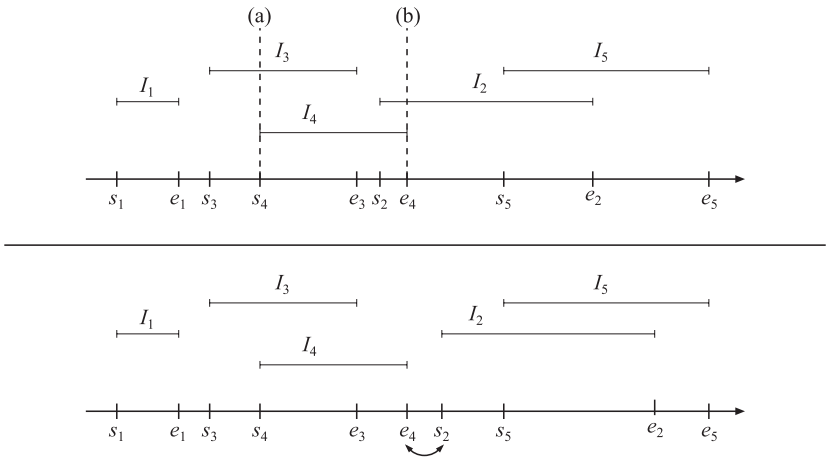

Figure 25.3. At the top, the interval I_4 is encountered (at the point marked (a)) when there is only one interval in the active list (I_3), so it is concluded that I_4 and I_3 overlap. When I_2 is encountered, I_4 is still in the active list (since e_4 has not been encountered yet), and so I_4 and I_2 also overlap. When e_4 is encountered, I_4 is removed (at the point marked (b)) from the active list. At the bottom, I_2 has moved to the right, and when the insertion sort finds that s_2 and e_4 need to change places, it can also be concluded that I_2 and I_4 do not overlap any longer. *(Illustration after Witkin et al. [93].)*

This procedure would take $O(n \log n)$ to sort all the intervals, plus $O(n)$ to sweep the list and $O(k)$ to report k overlapping intervals, resulting in an $O(n \log n + k)$ algorithm. However, due to temporal coherence, the lists are not expected to change much from frame to frame, and so a *bubble sort* or *insertion sort* [48] can be used with great efficiency after the first pass has taken place. These sorting algorithms sort nearly-sorted lists in an expected time of $O(n)$.

Insertion sort works by building up the sorted sequence incrementally. We start with the first number in the list. If we consider only this entry, then the list is sorted. Next, we add the second entry. If the second entry is smaller than the first, then we change places of the first and the second entries; otherwise, we leave them be. We continue to add entries, and we change the places of the entries, until the list is sorted. This procedure is repeated for all objects that we want to sort, and the result is a sorted list.

To use temporal coherence to our advantage, we keep a boolean for each possible pair of intervals. For large models, this may not be practical, as it implies an $O(n^2)$ storage cost. Instead, a hash map could be used to work around this problem. A specific boolean is `TRUE` if the pair overlaps and `FALSE` otherwise. The values of the booleans are initialized at the first step of the algorithm when the first sorting is done. When the status of an interval pair changes, the boolean is inverted. This is also illustrated in Figure 25.3.

We can create a sorted list of intervals for all three main axes and use the preceding algorithm to find overlapping intervals for each axis. If all three intervals for a pair overlap, their AABBs (which the intervals represent) also overlap; otherwise, they do not. The expected time is linear, which results in an $O(n + k)$-expected-time sweep-and-prune algorithm, where, again, k is the number of pairwise overlaps. If all an object pair overlaps on all three axes (i.e., all three booleans are true), then that pair could be added to a list of colliding pairs for fast access by later stages. This makes for fast overlap detection of the AABBs. Note that this algorithm can deteriorate to the worst expected performance for the sorting algorithm, which may be $O(n^2)$. This can happen when clumping occurs. A common example is when a large number of objects are lying on a floor. If the z-axis is pointing in the normal direction of the floor, we get clumping on the z-axis. One solution would be to skip the z-axis entirely and only perform the testing on the x- and y-axes [25]. In many cases, this works well. Liu et al. [63] present a parallel version of SAP, where the sweep direction is optimized as well. Their algorithm targets the GPU, and there is code online.

25.1.2 Grids

While grids and hierarchical grids are best known as data structures for ray tracing acceleration, they can also be used for broad phase collision detection [88]. In its simplest form, a grid is just an n-dimensional array of non-overlapping grid cells that cover the entire space of the scene. Each cell is therefore a box, where all boxes have the same size. From a high level, broad phase CD with a grid starts with insertion of all the objects' BVs in our scene into the grid. Then, if two objects are associated with the same grid cell, we immediately know that the BVs of these two objects are likely to overlap. Hence, we perform a simple BV/BV overlap test, and if they collide, we can proceed to the second level of the CD system. To the left in Figure 25.4, a two-dimensional grid with four objects is shown.

To obtain good performance, it is important to choose a reasonable grid cell size. This problem is illustrated to the right in Figure 25.4. One idea is to find the largest object in the scene, and make the grid cell size large enough to fit that object at all possible orientations [25]. In this way, all objects will overlap at most eight cells in the case of a three-dimensional grid.

Storing a large grid can be quite wasteful, especially if significant portions of the grid are left unused. Therefore, it has been suggested that spatial hashing could be used instead [25, 86, 88]. In general, each grid cell is mapped to an index in a hash table. All objects overlapping with a grid cell are inserted into the hash table, and testing can continue as usual. Without spatial hashing, we need to determine the size of the AABB enclosing the entire scene before memory can be allocated for the grid. We also must limit where objects can move, so they do not leave these bounds. These problems are avoided altogether with spatial hashing, as objects can immediately be inserted into the hash table, no matter where they are located.

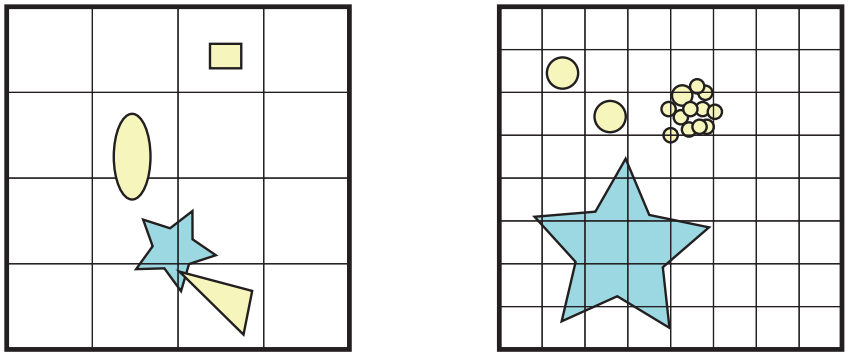

Figure 25.4. Left: a low-resolution two-dimensional grid with four objects. Note that since the ellipse and the star overlap a shared grid cell, these objects have to be tested using a BV/BV test. In this case, they do not collide. The triangle and the star also overlap a shared grid cell, and they do in fact collide. Right: a higher-resolution grid. As can be seen, the star overlaps with many grid cells, which can make this procedure expensive. Also note that there are many smaller objects where clumping occurs, and this is another possible source of inefficiency.

As illustrated in Figure 25.4, it is not always optimal to use the same grid cell size in the entire grid. Another option is to use hierarchical grids. In this scheme several different nested grids with different cell sizes are used, and an object is inserted in only the grid where the object's BV is (just) smaller than the grid cell. If the difference in grid cell size between adjacent levels in the hierarchical grid is exactly two, this structure is quite similar to an octree. There are many details worth knowing about when implementing grids and hierarchical grids for CD, and we refer to the excellent treatment of this topic by Ericson [25] and the article by Pouchol et al. [76].

25.1.3 Bounding Volume Hierarchies

Bounding volume hierarchies (BVHs) were described in Section 19.1.1 as a means to accelerate rendering using view frustum culling. How to build a BVH and how to perform collision testing using these are described in detail in Section 25.2. The broad phase can also be implemented using a BVH. Assume objects move every frame and that an AABB exists for each object in a scene, i.e., each leaf node contains an AABB and its underlying object. The BVH can be built rapidly using these AABBs. The maximum number of children is usually adjusted to fit the underlying target architecture, e.g., using 8 children if the target SIMD width is 8.

After the BVH has been built, it is possible to detect which object pairs overlap, i.e., which leaf nodes' AABBs overlap. One method is to hierarchically test each object's AABB against the BVH. If the object's AABB does not overlap a node's BV during traversing the BVH, then processing is terminated for that node's subtree. If instead there is an overlap, the recursion continues down. To find out what D, in Figure 25.5, overlaps, we start testing at the root, which it will always overlap. It

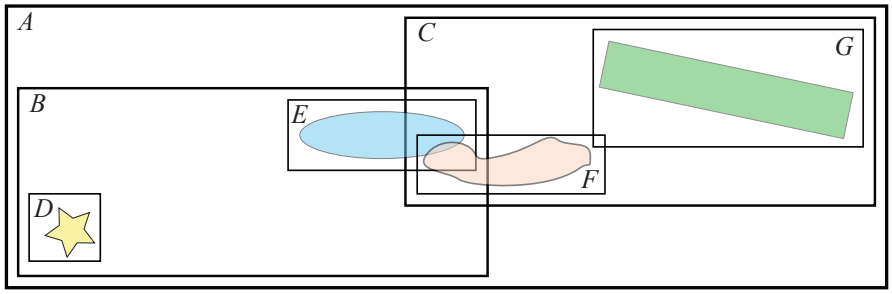

Figure 25.5. In this BVH, consisting of axis-aligned bounding boxes, there are 7 nodes, denoted A through G. A is the root and D-G are leaves, while B and C are internal nodes. See the text for two different methods to detect which leaf nodes overlap.

does not overlap C, so no further testing is needed in that subtree. Testing continues in B, which D always overlap since it is a child in B's subtree. Finally, D is tested against E, which indicates that D does not overlap with any other object. Next, we determine if E overlaps any other object. It overlaps A and B since E is in their subtrees, but it also overlaps C. Traversing down in B, we discover that E does not overlap anything there. In C, however, we will find that E overlaps F but not G. With this method, we test a leaf node X, and if recursion reaches a leaf node whose AABB, Y, overlaps with X then the pair (X, Y) is added to a list of overlapping pairs of objects. Note that some care has to be taken so that, for example, we do not add both (E, F) and (F, E) to the list.

Alternatively, one can make a BVH versus BVH test, where the BVH is tested against itself and recursion is terminated when two AABBs do not overlap. If a pair of leaf nodes' AABBs (corresponding to two different objects) are found to overlap, then this pair is added to a list which is then fed to the next phase. Note that care has to be taken so that we do not report overlap between an AABB and itself. Using Figure 25.5, testing starts at the root, and since a node will always overlap itself, we descend into A and process B and C. B overlaps itself, and we descend into B only to find that D and E do not overlap. We also descend into C, and find that F and G overlap, so (F, G) is added to the list of overlapping pairs. Next, since B and C overlap, we must test all B's children against all of C's children. Here, we find that E and F overlap, and add that pair to the list as well.

An important advantage of approaches using a BVH is that we build only a single data structure over the objects' AABBs, and this BVH can then be used for broad phase CD, view frustum culling, occlusion culling, and ray tracing, for example.

25.2 Mid Phase Collision Detection

After the broad phase, we assume that two objects are processed and that we want to find overlapping parts of these objects. To that end, this section describes general, hierarchical bounding volume collision detection algorithms. The models can undergo rigid-body motion, i.e., rotation plus translation, or even more general types of deformation (Section 25.7). These methods provide efficient bounding volumes (BVs), in that they try to create tight fitting volumes for a set of geometry. Smaller BVs improve the performance of these algorithms. Note also that meshes simplified from the original meshes used for rendering are often used to reduce the complexity and the cost of CD [73]. For simple colliders—such as boxes, spheres, and capsules—processing can skip ahead to the narrow phase.

This section will present some general ideas and methods for detecting collisions between two given models using bounding volume hierarchies (BVHs). Two elements shared by these algorithms are that they build a hierarchical representation of each model using bounding volumes, and that the high-level code for a collision query is similar, regardless of the kind of BV being used.

25.2.1 BVH Building

Initially, a model is represented by several primitives, which we assume are triangles in this section, but in general, they can be any type of primitive. Since each model should be represented as a hierarchy of some kind of bounding volumes, methods must be developed that build such hierarchies with the desired properties. A hierarchy that is commonly used in the case of collision detection algorithms is a data structure called a k-ary tree, where each node may have at most k children (Section 19.1). Many algorithms use the simplest instance of the k-ary tree, namely the binary tree, where $k = 2$. However, with current architectures, a higher value of k may be preferred (e.g., to match a particular SIMD width, or to minimize pointer indirection). At each internal node, there is a BV that encloses all its children in its volume, and at each leaf are one or more primitives. The bounding volume of an arbitrary node (either an internal node or a leaf), A, is denoted A_{BV}, and the set of children belonging to A is denoted A_c.

There are four main ways to build a hierarchy: a *bottom-up* method, an *incremental tree-insertion*, a *top-down* approach, or a *linear BVH* method. To create efficient, tight structures, the areas or the volumes of the BVs typically are minimized wherever possible [31, 47]. A problem with using the volume can be seen with a floor object, for example. The floor has no volume, so would have no size or influence by itself when using this metric. All single axis aligned polygons would then have the same influence, i.e., none at all, regardless of size. There is evidence showing that surface area is the better heuristic most of the time, while volume can give better performance when

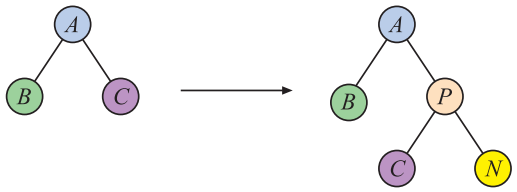

Figure 25.6. On the left, a binary tree with three nodes is shown. We assume that only one primitive is stored at each leaf. Now we wish to insert a new node, named N. This node has a bounding volume (BV) and a primitive that BV encloses. Therefore, the node will be inserted as a leaf node. In this example, say we find that the total tree volume is smaller if we insert N at the node called C (rather than at B). Then a new parent node P is created that encloses both C and the new node N.

objects with large surfaces are facing each other [95].[1] As such, we will use surface area for the remainder of this chapter, but recall that sometimes volume can be better.

The first of these methods, bottom-up, starts by combining several primitives and finding a BV for them. These primitives should be located close together, which can be determined by using the distance between the primitives. Then, either new BVs can be created in the same way, or existing BVs can be grouped with one or more BVs constructed in a similar way, thus yielding a new, larger parent BV. This is repeated until only one BV exists, which then becomes the root of the hierarchy. In this way, closely located primitives are always located near each other in the bounding volume hierarchy.

The incremental tree-insertion method starts with an empty tree. Then all other primitives and their BVs are added one at a time to this tree. This is illustrated in Figure 25.6. To make an efficient tree, an insertion point in the tree must be found. This point should be selected so that the total tree volume increase is minimized. A simple method for doing this is to descend to the child that gives a smaller increase in the tree. This kind of algorithm typically takes $O(n \log n)$ time. Randomizing the ordering in which primitives are inserted can improve tree formation [31].

The top-down approach, which is a popular method, starts by finding a BV for all primitives of the model, which then acts as the root of the tree. Then a divide-and-conquer strategy is applied, where the BV is first split into k or fewer parts. For each such part, all included primitives are found, and then a BV is created in the same manner as for the root, i.e., the hierarchy is created recursively. It is most common to find some axis along which the primitives should be split, and then to find a good split point on this axis. In Section 22.4 geometric probability is discussed, and this can be used when searching for a good split point, which will result in better BVHs. In ray tracing, the surface area heuristic (SAH) is used, which is formulated as

$$C(n) = \begin{cases} C_i A(n) + C(n_l) + C(n_r), & n \in I \\ C_t A(n) N(n), & n \in L, \end{cases} \tag{25.2}$$

[1] In writing this chapter, we thought about this question and believe the chance of two objects intersecting is relative to the surface area of the Minkowski difference, and so depends on both objects.

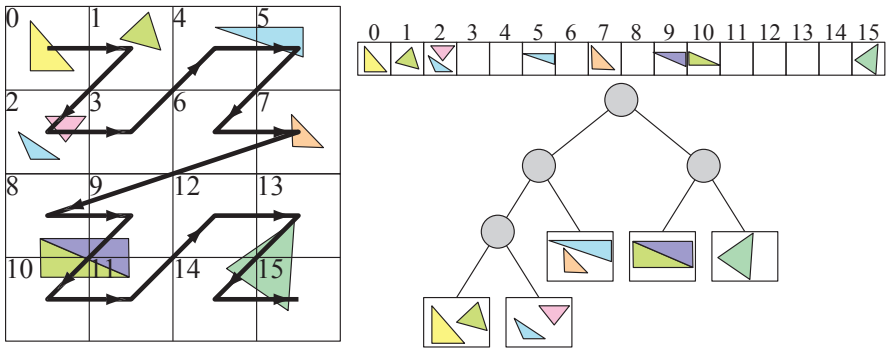

Figure 25.7. Left: the Morton curve is shown as the fat, Z-shaped curve. A triangle is assigned to the Morton code, whose center of the cells overlap the centroid of the triangle. The triangles are sorted according to the Morton codes (upper right). The final tree is shown to the lower left, which is created by splitting the sorted list hierarchically.

where $C(n)$ is the SAH cost of a node n, and n_l and n_r are the left and right children of n. The surface area of the bounding volume n is denoted $A(n)$, and C_i is the cost for traversing an internal node, while C_t is the cost of ray/triangle intersection. Also, the set of all internal nodes are denoted I and all leaf nodes are denoted L. $N(n)$ is the number of triangles in a leaf node. Note that the first line in the equation above dictates cost of making n an internal node and the second line is the cost of making it a leaf node. Ideally, the first line of the equation should be minimized and this is done by searching along the $x-$, $y-$, and z-axes for the best split point. Embree [90] has many state-of-the-art methods implemented for this. Also, at each point, one chooses either to build an internal node or make a leaf node based on which is least expensive. Since area is used in the SAH metric, it is not ideally suited for collision detection. However, the function $A(n)$ can be replaced for $V(n)$, which computes volume instead.

Note that a potential advantage of the top-down approach is that a hierarchy can be created lazily, i.e., on an as-needed basis. This means that we construct the hierarchy only for those parts of the scene where it is actually needed. But since this building process is performed during runtime, whenever a part of the hierarchy is created, the performance may go down significantly. This is not acceptable for applications such as games with real-time requirements, but may be a great time and memory saver for CAD applications or offline calculations such as path planning, animation, and more.

The linear BVH method [59] was originally targeting building BVHs on the GPU, but excels on CPUs as well. The core idea is illustrated in Figure 25.7. First a box is found around all triangles and at that point the triangles are stored in an array in some order. Next, a Morton code (Section 23.8), which is an integer, is assigned to each triangle based on its centroid. After that the triangles are sorted according to their Morton codes. Finally, the list is split hierarchically, and at each step an internal node

is created that bounds its triangles. This process stops when there are no triangles left in a sublist or when there are a small number that can fit in a leaf. Karras and Aila [45, 46] provide an overview of the many research papers on this topic and present highly optimized versions that works for BVHs, octrees, and kd-trees based on quickly building the BVH by interpreting the sorted Morton codes as a radix tree. Apetrei [1] improves on Karras' technique by merging two of the passes into a single pass while generating the same tree.

One challenge for CD algorithms is to find tight-fitting bounding volumes and hierarchy construction methods that create balanced and efficient trees. Note that balanced trees have better worst-case execution, since the depth of every leaf is the same (or almost the same). This means that it takes an equal amount of time to traverse the hierarchy down to any leaf (i.e., a primitive), and that the time of a collision query will not vary depending on which leaves are accessed. In this sense, a balanced tree is optimal. However, this does not mean that it is best for all inputs. For example, if part of a model will seldom or never be queried for a collision, then those parts can be located deep in an unbalanced tree, so that the parts that are queried most often are closer to the root [34]. The details of this procedure for an OBB tree is described on page 15.

Note also that several spatial data structures are described in relation to acceleration algorithms in Section 19.1, and that BV creation is treated in Section 22.3.

25.2.2 Collision Testing between BVHs

In the mid phase of CD, we will find out which leaves' BVs overlap with other leaves' BVs, i.e., this phase will create a list of leaf-leaf pairs whose BVs overlap. This information will be sent further to the narrow phase of CD (Section 25.3). Alternatively, overlapping BV pairs can have their contents immediately checked for collision using narrow phase testing.

A and B are two nodes in the model hierarchies, which at the first call are the roots of the models. A_{BV} and B_{BV} are used to access the BV of the appropriate node. Recall that A_c is the set of children nodes of A. The basic idea is to open a (larger) box when overlap is detected and recursively test its contents.

```
     MidPhaseCD(A, B)
1 :  if(isLeaf(A) and isLeaf(B))
2 :     add(A, B, L);  // add pair (A, B) to list L
3 :  else if(isNotLeaf(A) and isNotLeaf(B))
4 :     if(overlap(A_BV, B_BV))
5 :        if(SurfaceArea(A) > SurfaceArea(B))
6 :           for each child C ∈ A_c
7 :              MidPhaseCD(C, B)
8 :        else
9 :           for each child C ∈ B_c
10 :             MidPhaseCD(A, C)
11 : else if(isLeaf(A) and isNotLeaf(B))
12 :    if(overlap(A_BV, B_BV))
13 :       for each child C ∈ B_c
14 :          MidPhaseCD(C, A)
15 : else
16 :    if(overlap(B_BV, A_BV))
17 :       for each child C ∈ A_c
18 :          MidPhaseCD(C, B)
```

As can be seen in this pseudocode, there are portions of code that could be shared, but it is presented like this to show how the algorithm works. Some lines deserve some attention. Lines 1–2 add a pair of leaf nodes (A, B) to a list, L, of all overlapping leaf node pairs. Lines 3–10 handle the case where both nodes are internal nodes. The consequence of the comparison SurfaceArea(A) > SurfaceArea(B) is that the node with the largest surface area is descended. The idea behind this test is to get the same tree traversal for the calls **MidPhaseCD**(A, B) and for **MidPhaseCD**(B, A), so that traversal becomes deterministic. This test is also important since it tends to give better performance, as the largest box is traversed first at each step.

Another idea is to alternate between descending A and B. Doing so avoids the surface area computation, and so could be faster. Alternatively, the surface area can be precomputed for rigid bodies, but this computation requires extra memory per node. Also, for many BVs the actual surface area need not be computed, since it suffices with a computation that preserves the "surface area order." As an example, it suffices to compare the radii for spheres. Note that the list L is typically sent to the narrow phase as a next step.

25.2.3 BVH Cost Function

The function t in Equation 25.3 below was first introduced (in a slightly different form, without the last term) as a framework for evaluating the performance of hierarchical BV structures in the context of acceleration algorithms for ray tracing [92]. It has since been used to evaluate the performance of CD algorithms as well [33], and it has

been augmented by the last term to include a new cost specific to some systems [47, 50] that might have a significant impact on performance. This cost results from the fact that if a model is undergoing a rigid-body motion, then its BV and parts or all of its hierarchy might have to be recomputed, depending on the motion and the choice of BV:

$$t = n_v c_v + n_p c_p + n_u c_u. \tag{25.3}$$

Here, n_v is the number of BV/BV overlap tests, c_v is the cost for a BV/BV overlap test, n_p is the number of primitive pairs tested for overlap, c_p is the cost for testing whether two primitives overlap, n_u is the number of BVs updated due to the model's motion, and c_u is the cost for updating a BV.

Creating a better hierarchical decomposition of a model would result in lower values of n_v, n_p, and n_u. Creating better methods for determining whether two BVs or two triangles overlap would lower c_v and c_p. However, these are often conflicting goals, because changing the type of BV in order to use a faster overlap test usually means that we get looser-fitting volumes.

Examples of different bounding volumes that have been used in the past are spheres [42], axis-aligned bounding boxes (AABBs) [5, 40], oriented bounding boxes (OBBs) [33], k-DOPs (discrete oriented polytopes) [47, 49, 96], and capsules [55]. Spheres are the fastest to transform, and the overlap test is also fast, but they provide quite a poor fit. AABBs normally provide a better fit and a fast overlap test, and they are a good choice if there is a large amount of axially aligned geometry in a model (as is the case in most architectural models). OBBs have a much better fit, but slower overlap tests. The fit of the k-DOPs are determined by the parameter k—higher values of k give a better fit, slower overlap testing, and poorer transform speed.

25.2.4 OBBTree

The OBBTree [33] is an example of a particular BVH structure used for mid phase collision detection, and here will present it in some detail, as it is representative of a typical collision detection system. This scheme was designed to perform especially well if *parallel close proximity*, where two surfaces are quite close and nearly parallel, is found during collision detection. These kinds of situations often occur in tolerance analysis and in virtual prototyping. An example is mounting the machine parts of an engine, where one would like to make sure that the parts can be assembled easily without needing to building them first.

Choice of Bounding Volume

This method uses the oriented bounding box (OBB) as the bounding volume. However, many concepts in this section also apply to other BVs, such as the AABB. One reason to use OBBs is that a tree of them converges better to the underlying geometry than AABBs and spheres. However, building a tree with AABBs is faster than with OBBs, so performance depends on the use case. OBB/OBB overlap testing is treated in Section 22.13.5. Using the performance evaluation framework of Section 25.2.3,

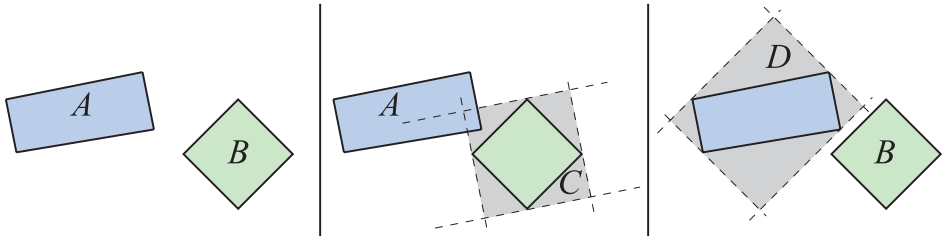

Figure 25.8. At the left are two OBBs (A and B) that are to be tested for overlap with an OBB/OBB overlap test that omits the last nine axis tests. If this is interpreted geometrically, then the OBB/OBB overlap test is approximated with two AABB-AABB overlap tests. The middle illustration shows how B is bounded by an AABB in the coordinate system of A. C and A overlap, so the test continues on the right. Here, A is enclosed with an AABB in the coordinate system of B. B and D are reported not to overlap. So, testing ends here. However, in three dimensions, D and B could overlap as well, and still A and B need not overlap due to the remaining axis tests. In such cases, A and B would erroneously be reported to overlap.

the preceding reasoning means that n_v and n_p are lower for OBBs than for AABBs and spheres.

Van den Bergen has suggested a simple technique for speeding up the overlap test between two OBBs [5, 7], where the last nine axis tests that correspond to a direction perpendicular to one edge of the first OBB and one edge of the second OBB are skipped. This test is often referred to as *SAT lite*. Geometrically, this can be thought of as doing two AABB/AABB tests, where the first test is done in the coordinate system of the first OBB and the other is done in the coordinate system of the other. This is illustrated in Figure 25.8. The shortened OBB/OBB test (which omits the last nine axis tests) will sometimes report two disjoint OBBs as overlapping. In such cases, the recursion in the OBBTree will go deeper than necessary. In testing, the net result was that the average performance was improved by skipping these tests. Van den Bergen's technique has been implemented in a collision detection package called SOLID [5, 6, 7, 9], which also handles deformable objects.

Hierarchy Building

The basic data structure of the OBBTree is an k-ary tree, i.e., an internal node may have up to k children, where each internal node holds an OBB (Section 22.2) and each external (leaf) node holds one or more triangles. The value k is usually chosen to fit with the target architecture. For example, one may choose $k = 8$ if the implementation is using AVX, which has a SIMD width of 8 for 32-bit floating point. The top-down approach developed by Gottschalk et al. for creating the hierarchy divided into finding a tight-fitting OBB for a triangle soup and then splitting this along one axis of the OBB, which also categorizes the triangles into two groups. For each of these groups, a new OBB is computed. The creation of OBBs is treated in Section 22.3.

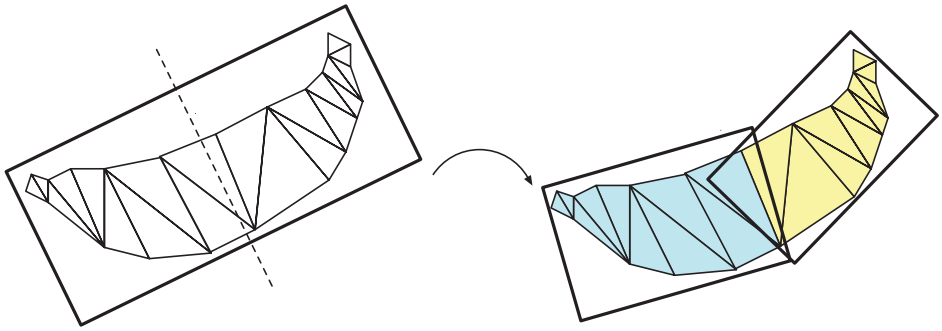

Figure 25.9. This figure shows how a set of geometry with its OBB is split along the longest axis of the OBB, at the split point marked by the dashed line. Then the geometry is partitioned into two subgroups, and an OBB is found for each of them. This procedure is repeated recursively to build the OBBTree.

After we have computed an OBB for a set of triangles, the volume and the triangles should be split and formed into two new OBBs. Gottschalk et al. use a strategy that takes the longest axis of the box and splits it into two parts of the same length. An illustration of this procedure is shown in Figure 25.9. A plane that splits the box into two equal parts is used to partition the triangles into two subgroups. A triangle that crosses this plane is assigned to the group that contains its centroid. In the rare case that all centroids of all triangles are located on the dividing plane, or that all centroids are located on the same side of the splitting plane, the other axes are tried in diminishing order. Note that using a surface area heuristic (Equation 25.2) is likely to yield better BVHs [95].

For each of the subgroups, the matrix method briefly presented in Section 22.3 is used to compute (sub-) OBBs. If the OBB is instead split at the median center point, where each child has the same number of triangles, then balanced trees are obtained. In the related field of ray tracing there is extensive research on other splitting strategies [89].

Handling Rigid-Body Motions

In the OBBTree hierarchy, each OBB, A, is stored together with a rigid-body transformation (a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$) matrix $\mathbf{M}_A$. This matrix holds the relative orientation and position of the OBB with respect to its parent.

Now, assume that we start to test two OBBs, A and B, against each other. The overlap test between A and B should then be done in the coordinate system of one of the OBBs. Let us say that we decide to do the test in A's coordinate system. In this way, A is an AABB (in its own coordinate system) centered around the origin. The idea is then to transform B into A's coordinate system. This is done with the matrix below, which first transforms B into its own position and orientation (with $\mathbf{M}_B$) and then into A's coordinate system (with the inverse of A's transform, $\mathbf{M}_A^{-1}$).

Recall that for rigid body transforms, the transpose is the inverse, so little additional computation is needed:

$$\mathbf{T}_{AB} = \mathbf{M}_A^{-1}\mathbf{M}_B. \tag{25.4}$$

The OBB/OBB overlap test takes as input a matrix consisting of a 3×3 rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$, which hold the orientation and position of B with respect to A (Section 22.13.5), so $\mathbf{T}_{AB}$ is decomposed as below:

$$\mathbf{T}_{AB} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 0 \end{pmatrix}. \tag{25.5}$$

Now, assume that A and B overlap, and that we want to descend into A's child called C. This can be done as follows. We choose to do the test in C's coordinate system. The idea is then to transform B into A's coordinate system (with $\mathbf{T}_{AB}$) and then transform that into the coordinate system of C (using $\mathbf{M}_C^{-1}$). This is done with the following matrix, which is then used as the input to the OBB/OBB overlap test:

$$\mathbf{T}_{CB} = \mathbf{M}_C^{-1}\mathbf{T}_{AB}. \tag{25.6}$$

This procedure is then used recursively to test all OBBs.

Miscellaneous

The **MidPhaseCD** pseudocode for collision detection between two hierarchical trees, as presented in Section 25.2.2, can be used on two trees created with the preceding algorithms. All that needs to be exchanged is the `overlap()` function, which should point to a routine that tests two OBBs for overlap.

All algorithms involved in OBBTree have been implemented in a free software package called *RAPID* (Robust and Accurate Polygon Interference Detection) [33].

25.3 Narrow Phase Collision Detection

At this point, the broad and the mid phase has narrowed down the data to a list L containing pairs of leaf nodes whose BVs overlap. The goal here is to either compute all pairs of intersecting primitives, to assist in avoiding objects that penetrate each other, or to use distance queries to determine if objects are sufficiently far from each other. These two topics are described next.

25.3.1 Primitive versus Primitive

In general, we have a list, L, of pairs of leaf nodes that overlap when entering this phase. In the simplest implementation one may perform a loop over each primitive in the first leaf and test against all primitives in the other leaf. Primitive-primitive tests are described in Chapter 22.

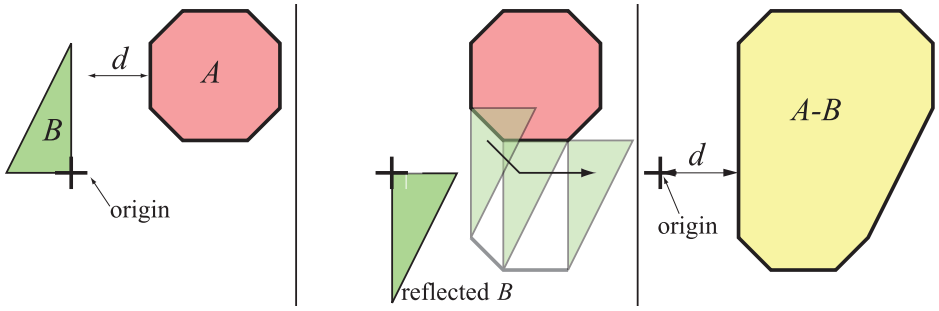

Figure 25.10. To the left, two convex objects A and B are shown. To construct $A - B$, first move A and B so that one reference point is at the origin (already done at the left). Then B is reflected, as shown in the middle, and the chosen reference point on B is put on the surface of A and then the reflected B is swept around A. This creates $A - B$, to the right. The minimum distance, d, is shown both on the left and the right.

It should be noted that, from a code perspective, it may or may not be advantageous to merge the mid and narrow phases to a single phase. A merge might give simpler code, but with a single list of all pairs, one can sort so that all types of primitive-primitive pairs are adjacent. This sorting can make for a faster parallel SIMD implementation.

When the pair list, L, contains convex polyhedrons, one can use distance queries to determine if two objects penetrate and, if so, move them apart until they do not. This is the topic of the next section.

25.3.2 Distance Queries

In certain applications one wants to test whether an object is at least a certain distance from an environment. For example, when designing a new car, there has to be enough space for different-sized passengers to be able to seat themselves comfortably. Therefore, a virtual human of various sizes can be tried in the car's seats, to see if they can be seated without bumping into the car. Preferably, the passengers should be able to seat themselves and still be at least, say, 10 cm apart from some of the car's interior elements. This sort of testing is called *tolerance verification*. This can be used for *path planning*, i.e., how an object's collision-free path from one point to another can be determined algorithmically. Given the acceleration and velocity of an object, the minimum distance can be used to estimate a lower bound on the time to impact. In this way, collision detection can be avoided until that time [62]. Another related query is of the penetration depth, that is, finding out how far two objects have moved into each other. This distance can be used to move the objects back just enough so that they do not penetrate any longer, and then an appropriate collision response can be computed at that point.

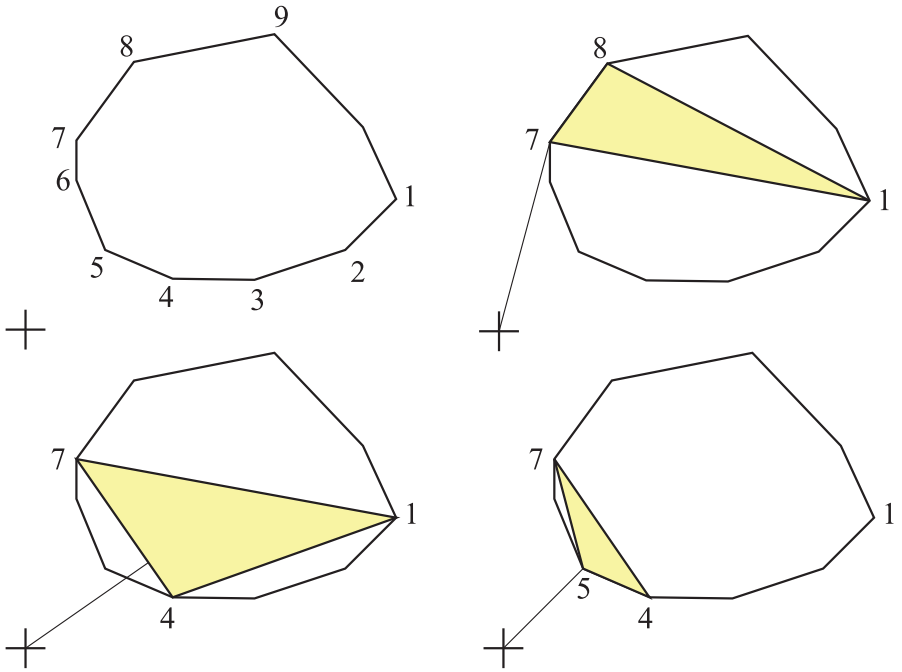

Figure 25.11. GJK. Upper left: The minimum distance between the origin and the polygon is to be computed. Upper right: An arbitrary triangle is chosen as a starting point for the algorithm, and the minimum distance to that triangle is computed. Vertex 7 is closest. Lower left: In the next step (not shown), all vertices are projected onto the line from the origin to vertex 7, and the closest vertex replaces the one in the triangle that is farthest away on this projection. Thus vertex 4 replaces vertex 8. The closest point on this triangle is then found, which is located on the edge from vertex 4 to 7. Lower right: The vertices are projected onto the line from the origin to the closest point from the previous step, and vertex 5 thus replaces vertex 1, which is farthest away on the projection. Vertex 5 is the closest point on this triangle, and when the vertices are projected on the line to vertex 5, we find that vertex 5 is the closest point overall. This completes the iteration. At this time the closest point on the triangle is found, which also happens to be vertex 5. This point is returned. *(Illustration after Jiménez et al. [44].)*

One of the first practical approaches developed that compute the minimum distance between convex polyhedra is called *GJK*, after its inventors Gilbert, Johnson, and Keerthi [30]. An overview of this algorithm is given in this section. GJK computes the minimum distance between two convex objects, A and B. To do this, the *difference object* (sometimes called the *sum object*) between A and B is used [6]:

$$A - B = \{\mathbf{x} - \mathbf{y} : \mathbf{x} \in A, \mathbf{y} \in B\}. \tag{25.7}$$

This is also called the *Minkowski sum* of A and (reflected) B (Section 25.11.3). All differences $\mathbf{x} - \mathbf{y}$ are treated as a point set, which forms a convex object. An example of such a difference is shown in Figure 25.10.

The idea of GJK is that, instead of computing the minimum distance between A and B, we calculate the minimum distance between $A - B$ and the origin. These two distances can be shown to be equivalent. The algorithm is visualized in Figure 25.11. Note that if the origin is inside $A - B$ then A and B overlap.

The algorithm starts from an arbitrary simplex in the polyhedron. A simplex is the simplest primitive in the respective dimension, so it is a triangle in two dimensions, and a tetrahedron in three dimensions. This starting element can be any valid simplex, e.g., any tetrahedron fully inside our polyhedron. Then the point on this simplex closest to the origin is computed. Van den Bergen shows how this can be done by solving a set of linear equations [6, 7]. A vector is then formed starting at the origin and ending at the nearest point. All vertices of the polyhedron are projected onto this vector, and the one with the smallest projection distance from the origin is chosen to be a new vertex in the updated simplex. Since a new vertex will be added to the simplex, an existing vertex in the simplex must then be removed (else it would not remain a simplex). The point whose projection is farthest away is deleted. Once this procedure is done, the minimum distance to the updated simplex is computed, and the algorithm iterates through all vertices again until the algorithm cannot update the simplex any longer. The algorithm terminates in a finite number of steps for two polyhedra [30]. The performance of this algorithm can be improved using many techniques, such as incremental computation and caching [6].

Van den Bergen describes a fast and robust implementation of GJK [6, 7]. GJK can also be extended to compute penetration depth [8, 12, 37]. In particular, van den Bergen [8, 9] describes a method called the *expanding polytope algorithm* (EPA), which is based on GJK. When penetration has occurred, the origin (represented by a plus-sign) in Figure 25.11 is found to be inside the polygon. EPA then performs similar steps as GJK to find the point on the polygon that is closest to the origin. There are several other algorithms computing minimum distance, e.g., such as the Lin-Canny algorithm [61], V-Clip [70], PQP [55] SWIFT [23], and SWIFT++ [24], which also computes distance between concave rigid bodies.

It is often beneficial to use the separating axis from the previous frame as the starting vector when computing GJK for the current frame [8]. When temporal coherence is high, the algorithm can often terminate in the first step and Gregorius [37] reports an order of magnitude speedup. Note that it is also possible to use a method based on tracing rays to compute penetration distance. See Figure 25.13 in the next section.

When using the separating axis theorem on convex polyhedra to find penetration depth, another optimization is to store a simpler object, e.g., a sphere, fully inside each convex polyhedron and use it to cull axis tests [85]. The amount of overlap of two spheres on a separating axis will always be less than or equal to the amount of overlap for the two convex polyhedra. Our goal is to find the minimum penetration depth, the minimum we need to move two objects to separate them. Assuming that after some axis tests the current minimum penetration depth is d, then we do not need to further care about the current axis if the amount of overlap of the spheres is $> d$, because then the amount of overlap of the convex polyhedra will be even larger. This can result in a substantial increase in performance.

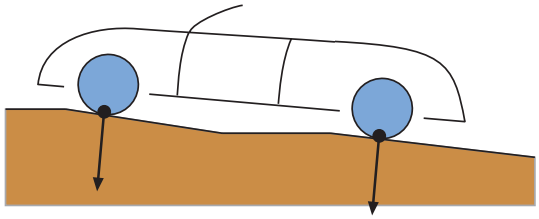

Figure 25.12. Instead of computing the collision between the entire car and the environment (the road), we place a ray at each wheel. These rays are then tested for intersection against the environment.

Note that when distance queries are used, no triangle-triangle intersections need to be computed since we instead obtain minimum distances or minimum penetration depths directly.

25.4 Collision Detection with Rays

In this section, we will present a fast technique that works well under certain circumstances. Imagine that a car drives upward on an inclined road and that we want to use the information about the road (i.e., the primitives of which the road is built) to steer the car upward. This could, of course, be done by testing all primitives of all car wheels against all primitives of the road, using the techniques from Sections 25.1 through 25.3. However, for games and some other applications, this kind of detailed collision detection is not always needed. Instead, we can approximate a moving object with a set of *rays*. In the case of the car, we can put one ray at each of the four wheels (see Figure 25.12). This approximation works well in practice, as long as we can assume that the four wheels are the only places of the car that will be in contact with the environment (the road). Assume that the car is standing on a plane at the beginning, and that we place each ray at a wheel so that each origin lies at the place where the wheel is in contact with the environment. The rays at the wheels are then intersection-tested against the environment. If the distance from a ray origin to the environment is zero, then that wheel is exactly on the ground. If the distance is greater than zero, then that wheel has no contact with the environment, and a negative distance means that the wheel has penetrated the environment. The application can use these distances for computing a collision response—a negative distance would move the car (at that wheel) upward, while a positive distance would move the car downward (unless the car is flying though the air for a short while). Note that this type of techniques could be harder to adapt to more complicated scenarios. For example, if the car crashes and is set in rotating motion, many more rays in different directions are needed.

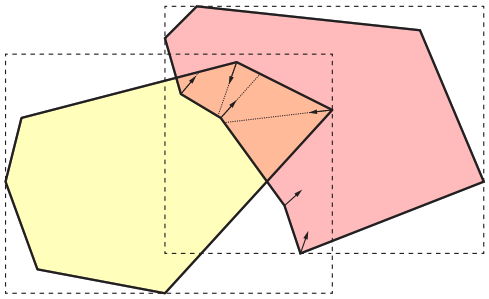

Figure 25.13. Vertices in the region where both boxes overlap shoot rays in their negative normal direction against the other object. The two lower right rays hits the same object as they originate from, and so are not processed further. A collision response is computed based on the lengths and directions of the dotted lines.

To speed up the intersection testing, we can use the same technique we most often use to increase efficiency in computer graphics—a *hierarchical representation*. The environment can be represented by an axis-aligned BSP tree (Section 19.l.2), bounding volume hierarchies, grids, (loose) octrees, or other data structures. For example, Frye [28] uses a loose octree (Section 19.1.3) for dynamic geometry and an axis-aligned BSP-tree for static geometry. Depending on what primitives are used in the environment, different ray/object intersection test methods are needed (Chapter 22).

Unlike standard ray tracing, where we need the closest object in front of the ray, what is actually desired is the intersection point furthest back along the ray, which can have a negative distance. To avoid having to treat the ray as searching in two directions, the test ray's origin is moved back until it is outside the bounding box surrounding the object, and is then tested against the environment. In practice, this just means that, instead of a ray starting at a distance 0, it starts at a negative distance that lets it begin outside the object's box. To handle a more general setting, such as driving a car through a tunnel and detecting collision against the roof, one would then have to search in both directions.

Hermann et al. [41] presents a different approach to using rays in collision detection. This is illustrated in Figure 25.13. When two overlapping boxes have been found, only the vertices in the overlapping region are processed. For each such vertex, a ray is shot in the negative vertex normal direction and tested only against the other object. If there is an intersection with the other object with a positive intersection distance, then the vertex and intersection point are kept as a collision pair. This test is added to Hermann et al. further describe how to filter out invalid pairs. The normals and collision pairs are then used to compute a collision response. For example, a penalty force can be applied in the direction of the vector between the two points in a collision pair and the size of the force made proportional to that vector's length. Lehericey et al. [60] exploit temporal coherence to provide a faster variant of this method.

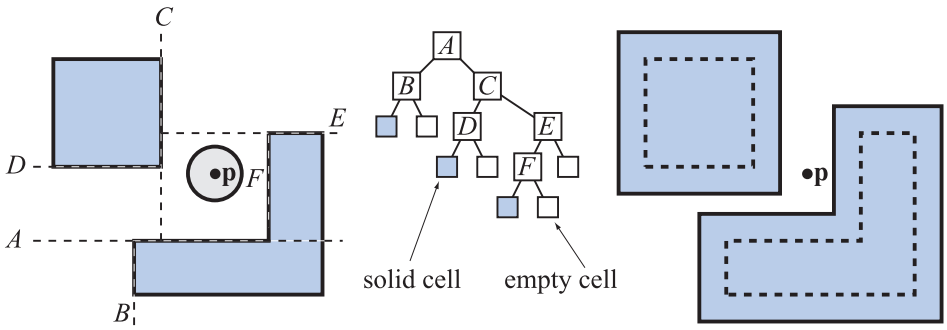

Figure 25.14. To the left is some geometry (blue) seen from above. Its BSP tree is shown in the middle. To test this tree against a circle with origin at $\mathbf{p}$, the BSP tree is grown outwards with the circle's radius, and then the point $\mathbf{p}$ can instead be tested against the grown BSP tree. This is shown to the right. Notice that the corners should be rounded, so this is an approximation that the algorithm introduces.

25.5 Dynamic CD Using BSP Trees

Here, the collision detection algorithm by Melax [65, 66] will be presented. It determines collisions between the geometry described by a BSP tree (Section 19.l.2), and a collider that can be either a sphere, a cylinder, or the convex hull of an object. It also allows for dynamic collision detection. For example, if a sphere moves from a position $\mathbf{p}_0$ at frame n to $\mathbf{p}_1$ at frame $n+1$, the algorithm can detect if a collision occurs anywhere along the straight line path from $\mathbf{p}_0$ to $\mathbf{p}_1$. The presented algorithm has been used in commercial games, where a character's geometry was approximated by a cylinder.

The standard BSP tree can be tested against a line segment quite efficiently. A line segment can represent a point (particle) that moves from $\mathbf{p}_0$ to $\mathbf{p}_1$. There may be several intersections, but the first one (if any) represents the collision between the point and the geometry represented in the BSP tree. Note that, in this case, the BSP tree is surface aligned, not axis-aligned. That is, each plane in the tree is coincident with a wall, floor, or ceiling in the scene. This is easily extended to handle a sphere, with radius r, that moves from $\mathbf{p}_0$ to $\mathbf{p}_1$ instead of a point. Instead of testing the line segment against the planes in the BSP tree nodes, each plane is moved a distance r along the plane normal direction. See Section 25.11 for similar ways of recasting intersection tests. This sort of plane adjustment is illustrated in Figure 25.14.

This is done for every collision query on the fly, so that one BSP tree can be used for spheres of any size. Assuming a plane is $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$, the adjusted plane is $\pi' : \mathbf{n} \cdot \mathbf{x} + d \pm r = 0$, where the sign of r depends on which side of the plane you continue testing/traversing in search of a collision. Assuming that the character is supposed to be in the positive half-space of the plane, i.e., where $\mathbf{n} \cdot \mathbf{x} + d \geq 0$, we would have to subtract the radius r from d. Note then that the negative half-space is considered "solid," i.e., something that the character cannot step through.

A sphere does not approximate a character in a game particularly well, but a few spheres may work well enough [42]. The convex hull of the vertices of a character or a cylinder surrounding the character does a better job. To use these other bounding volumes, d in the plane equation has to be adjusted differently. To test a moving convex hull of a set of vertices, S, against a BSP tree, the scalar value in Equation 25.8 below is added to the d-value of the plane equation [65]:

$$-\max_{\mathbf{v}_i \in S}(\mathbf{n} \cdot (\mathbf{v}_i - \mathbf{p}_0)). \tag{25.8}$$

The minus sign, again, assumes that characters are in the positive half-space of the planes. The point $\mathbf{p}_0$ can be any point found suitable to be used as a reference point. For the sphere, the center of the sphere was implicitly chosen. For a character, a point close to the feet may be chosen, or perhaps a point at the navel. Sometimes this choice simplifies equations (as the center of a sphere does). It is this point $\mathbf{p}_0$ that is tested against the adjusted BSP tree. For a dynamic query, that is, where the character moves from one point to another during one frame, this point $\mathbf{p}_0$ is used as the start point of the line segment. Assuming that the character is moved with a vector $\mathbf{w}$ during one frame, the endpoint of the line segment is $\mathbf{p}_1 = \mathbf{p}_0 + \mathbf{w}$.

The cylinder is perhaps even more useful because it is faster to test and still approximates a character in a game fairly well. However, the derivation of the value that adjusts the plane equation is more involved. What we do in general for this algorithm is that we recast the testing of a bounding volume (sphere, convex hull, and cylinder, in this case) against a BSP tree into testing a point, $\mathbf{p}_0$, against the adjusted BSP tree. This is the same as Minkowski sums (Section 25.11.3 and 25.3.2). Then, to extend this to a moving object, the point $\mathbf{p}_0$ is replaced by testing with a line segment from $\mathbf{p}_0$ to the destination point $\mathbf{p}_1$

We derive such a test for a cylinder, whose properties are shown to the top left in Figure 25.15, where the reference point, $\mathbf{p}_0$, is at the bottom center of the cylinder. Figure 25.15(b) shows what we want to solve: testing the cylinder against the plane π. In Figure 25.15(c), we move the plane π so that it barely touches the cylinder. The distance, e, from $\mathbf{p}_0$ to the moved plane is computed. This distance, e, is then used in Figure 25.15(d) to move the plane π into its new position π'. Thus, the test has been reduced to testing $\mathbf{p}_0$ against π'. The e-value is computed on the fly for each plane each frame. In practice, a vector is first computed from $\mathbf{p}_0$ to the point $\mathbf{t}$, where the moved plane touches the cylinder. This is shown in Figure 25.15(c). Next, e is computed as

$$e = |\mathbf{n} \cdot (\mathbf{t} - \mathbf{p}_0)|. \tag{25.9}$$

Now all that remains is to compute $\mathbf{t}$. The z-component (i.e., cylinder axis direction) of $\mathbf{t}$ is simple; if $n_z > 0$, then $t_z = p_{0z}$, i.e., the z-component of $\mathbf{p}_0$. Else, $t_z = p_{0z} + h$. These t_z values correspond to the bottom and top of the cylinder. If n_x and n_y are both zero (e.g., for a floor or ceiling), then we can use any point on the caps of the cylinder. A natural choice is $(t_x, t_y) = (p_x, p_y)$, the center of the cylinder

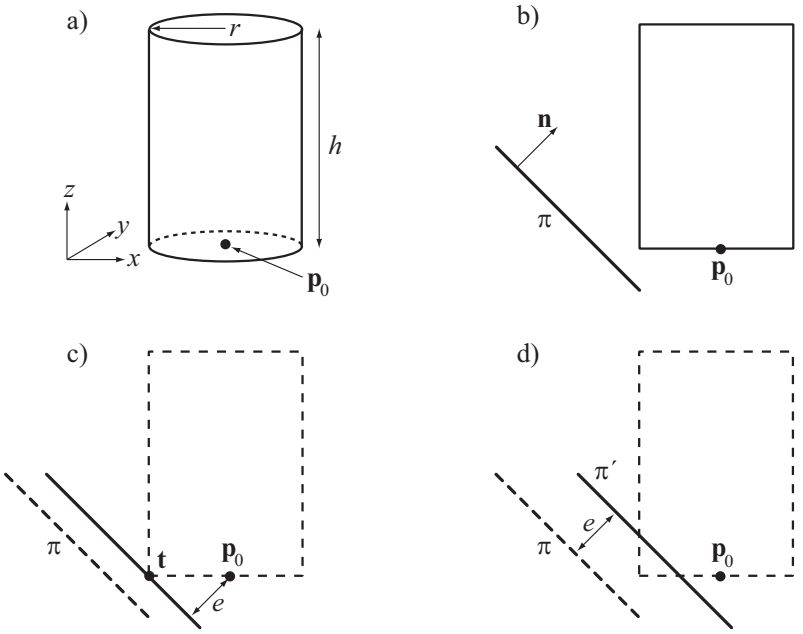

Figure 25.15. Figure (a) shows a cylinder with height h, radius r, and reference point $\mathbf{p}_0$. The sequence (b)-(d) then shows how testing a plane, π, against a cylinder (shown from the side) can be recast into testing a point $\mathbf{p}_0$ against a new plane π'. Since $\mathbf{p}_0$ is in the positive half-space of π', there is no overlap in this case.

cap. Otherwise, for a non-vertical $\mathbf{n}$, the following choice gives a point on the rim of the cylinder cap:

$$t_x = \frac{-rn_x}{\sqrt{n_x^2 + n_y^2}} + p_x, \qquad t_y = \frac{-rn_y}{\sqrt{n_x^2 + n_y^2}} + p_y. \tag{25.10}$$

That is, we project the plane normal onto the xy-plane, normalize it, and then scale by r to land on the rim of the cylinder.

Inaccuracies can occur using this method. One case is shown in Figure 25.16. As can be seen, this can be solved by introducing extra *bevel planes*. In practice, the "outer" angle between two neighboring planes are computed, and an extra plane is inserted if the angle is greater than 90°. The idea is to improve the approximation of what should be a rounded corner. In Figure 25.17, the difference between a normal BSP tree and a BSP tree augmented with bevel planes can be seen. The beveling planes certainly improves the accuracy, but it does not remove all errors.

Pseudocode for this collision detection algorithm follows below. It is called with the root N of the BSP tree, whose children are N.negativechild and N.positivechild,

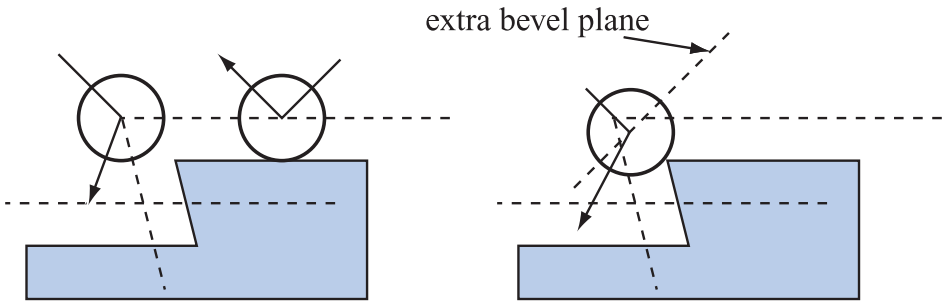

Figure 25.16. In the left illustration, the right sphere collides correctly, while the left sphere detects a collision too early. To the right, this is solved by introducing an extra bevel plane, which actually does not correspond to any real geometry. As can be seen the collision appears to be more correct using such planes. *(Illustration after Melax [65].)*

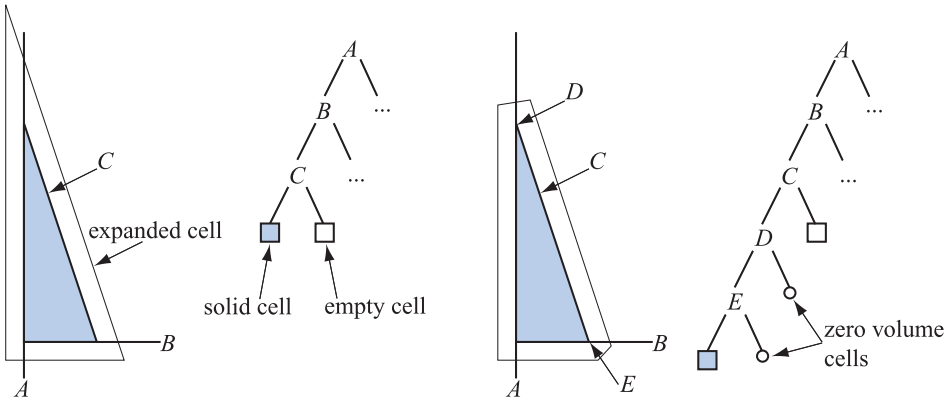

Figure 25.17. To the left, a normal cell and its BSP tree is shown. To the right, bevel planes have been added to the cell, and the changes in the BSP tree are shown. *(Illustration after Melax [65].)*

and the line segment defined by $\mathbf{p}_0$ and $\mathbf{p}_1$. Note that the point of impact (if any) is returned in a global variable called $\mathbf{p}_{\text{impact}}$:

```
     HitCheckBSP(N, p0, p1)
     returns ({TRUE, FALSE});
1 :  if(isEmptyLeaf(N)) return FALSE;
2 :  if(isSolidCell(N))
3 :     p_impact = p0
4 :     return TRUE;
5 :  end
6 :  hit = FALSE;
7 :  if(clipLineInside(N shift out, p0, p1, &w0, &w1))
8 :     hit = HitCheckBSP(N.negativechild, w0, w1);
9 :     if(hit) p1 = p_impact
10 : end
11 : if(clipLineOutside(N shift in, p0, p1, &w0, &w1))
12 :    hit| = HitCheckBSP(N.positivechild, w0, w1);
13 : end
14 : return hit;
```

The function `isSolidCell` returns `TRUE` if we have reached a leaf, and we are on the solid side (as opposed to the empty). See Figure 25.14 for an illustration of empty and solid cells. The function `clipLineInside` returns `TRUE` if part of the line segment (defined by the movement path $\mathbf{v}_0$ and $\mathbf{v}_1$) is inside the node's shifted plane, that is, in the negative half-space. It also clips the line against the node's shifted plane, and returns

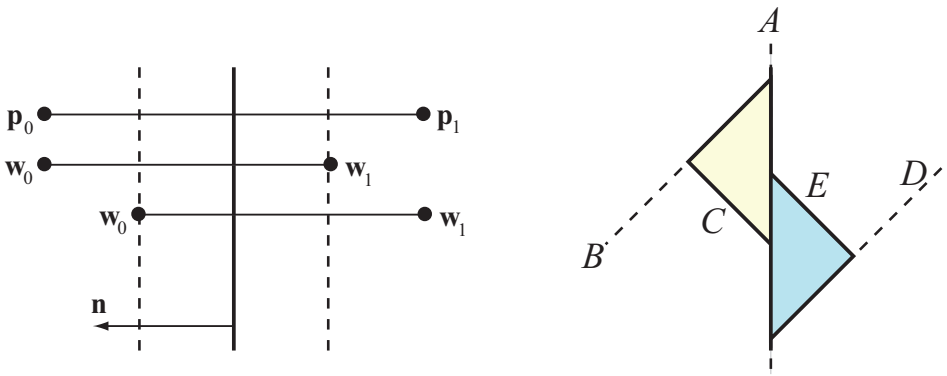

Figure 25.18. To the left, the line segment defined by $\mathbf{p}_0$ and $\mathbf{p}_1$ is clipped against a plane defined by a normal $\mathbf{n}$. The dynamic adjustments of this plane is shown as dashed lines. The functions `clipLineInside` and `clipLineOutside` returns the line segments defined by $\mathbf{w}_0$ and $\mathbf{w}_1$. Note that all three lines should have the same y-coordinates, but they are shown like this for clarity. To the right, an example is shown that explains why the lines should be clipped as shown to the left. The node A in the BSP tree belongs to both the triangle on the left and on the right. Therefore, it is necessary to move its plane in both directions.

the resulting line segment in $\mathbf{w}_0$ and $\mathbf{w}_1$. The function `clipLineOutside` is similar. Note also that the line segments returned by `clipLineInside` and `clipLineOutside` overlap each other. The reason for this is shown in Figure 25.18, as well as how the line is clipped. Line 9 sets $\mathbf{v}_1 = \mathbf{p}_{\text{impact}}$, and this is simply an optimization. If a hit has been found, and thus a potential impact point, $\mathbf{p}_{\text{impact}}$, then nothing beyond this point need to be tested since we want the first point of impact. On Lines 7 and 11, N is shifted "out" versus "in." These shifts refer to the adjusted plane equations derived earlier for spheres, convex hulls, and cylinders.

The advantage of this scheme is that only a single BSP tree is needed to test all characters and objects. The alternative would be to store different BSP trees for each different radius and object type. Other dynamic intersection tests are presented in Section 25.11.

25.6 Time-Critical Collision Detection

Assume that a certain game engine is rendering a scene in 14 ms when the viewer looks up at the sky, but that the rendering takes 30 ms when the viewer is looking toward the horizon. Clearly, this will give considerably different frame rates, which is usually disturbing to the user. One rendering algorithm that attempts to achieve constant frame rate is presented in Section 19.9.3. Here, another approach, called *time-critical collision detection*, is taken, which can be used if the application uses CD as well. It is called "time-critical" because the CD algorithm is given a certain time frame, say 9 ms, to complete its task, and it must finish within this time. Another reason to use such algorithms for CD is that it is critical for the perceived causality [74, 75], e.g., rapidly detecting whether one object causes another to move.

The following algorithm was introduced by Hubbard [42]. The idea is to traverse the bounding volume hierarchies in *breadth-first* order. This means that all nodes at one level in the tree are visited before descending to the next level. This is in contrast to *depth-first traversal*, which traverses the shortest way to the leaves (as done in the pseudocode in Section 25.2.2). These two traversals are illustrated in Figure 25.19. The reason for using breadth-first traversal is that both the left and the right subtree of a node are visited, which means that BVs which together enclose the entire object are visited. With depth-first traversal, we might only visit the left subtree because the algorithm might run out of time. When we do not know whether we have time to traverse the whole tree, it is at least better to traverse a little of both subtrees.

The algorithm first finds all pairs of objects whose BVs overlap, using, for example, the algorithm in Section 25.1. These pairs are put in a queue, called Q. The next phase starts by taking out the first BV pair from the queue. Their children BVs are tested against each other and if they overlap, the children pairs are put at the end of the queue. Then the testing continues with the next BV pair in the queue, until either the queue is empty (in which case all of the tree has been traversed) or until we run out of time [42].

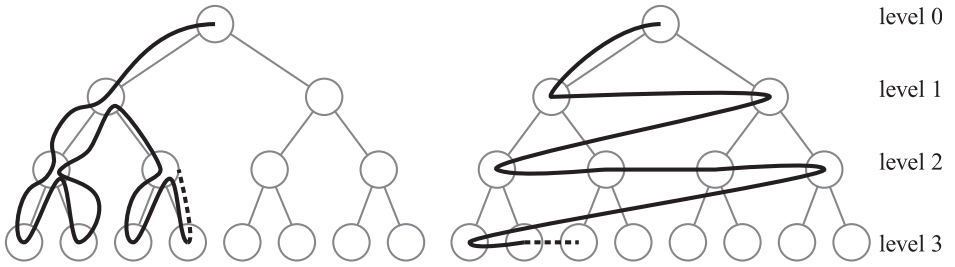

Figure 25.19. Depth-first (left) versus breadth-first (right) traversal. Depth-first traversal is often used in collision detection when traversing the bounding volume hierarchies, but for time-critical CD, breadth-first traversal is used.

Another related approach is to give each BV pair a priority and sort the queue on this priority. This priority can be based on factors such as visibility, eccentricity, and distance. Dingliana and O'Sullivan describe algorithms for computing approximate collision response and approximate collision contact determination [19]. This is needed for time-critical CD since the time may run out before the tree traversal has finished. Mendoza and O'Sullivan present a time-critical CD algorithm for deformable objects [67]. Kulpa et al. [51] performed user studies on using level of detail techniques, and provide a system where collision avoidance can be relaxed in the least perceptible manner.

25.7 Deformable Models

So far, the main focus of this section has been on either static models or rigid-body animated models. There are other sorts of motion, such as waves on the water or a piece of cloth swaying in the wind. This type of motion is generally not possible to describe using rigid bodies, and instead, one can treat each vertex as being an independent vector function over time. Collision detection for such models is generally more expensive.

Assuming that the mesh connectivity stays the same for an object during deformation, it is possible to design clever algorithms that exploit this property. Such deformation is what would happen to a piece of cloth in the wind (unless it is somehow torn apart). As a preprocess, an initial hierarchical bounding volume (BV) tree is built. Instead of actually rebuilding the tree when deformation has taken place, the bounding volumes are simply refitted to the deformed geometry [5, 56]. By using AABBs, which are fast to recompute, this operation is pretty efficient (as compared to OBBs). In addition, merging k children AABBs into a parent AABB is also quick and gives an optimal parent AABB. However, in general, any type of BV can be used. Van den Bergen organizes his tree so that all BVs are allocated and placed in an array, where a node always is placed so that its index is lower than its child nodes [5, 7].

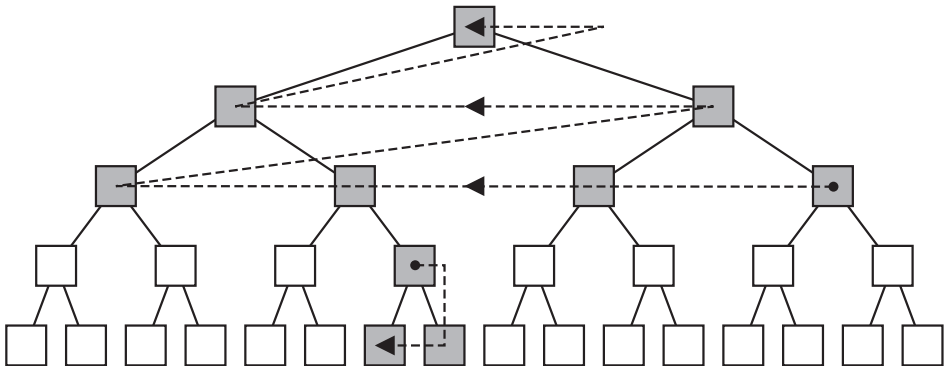

Figure 25.20. The hybrid bottom-up/top-down tree update method. The upper levels are updated using a bottom-up strategy, and only those nodes deeper down in the tree that are reached during a tree traversal are updated using a top-down method.

In this way, a bottom-up update can be done by traversing the array from the end backward, recomputing each BV at each node. This means that the BVs of the leaves are recomputed first, and then their parents' BVs are recomputed using the newly computed BVs, and so on back to the root of the tree. This sort of refit operation is reported to be about ten times as fast as rebuilding the tree from scratch [5].

However, it has been noted that, in general, few BVs in a tree need to be updated. This is because most of them are not used during a collision query. A hybrid bottom-up/top-down tree update has therefore been proposed [56]. The idea is to use a bottom-up update for the higher levels (including the root), which means that only the upper BVs will be updated each frame. The rationale for this is that the upper levels often prune away most geometry. These updated upper levels are tested for overlap with the other tree (also possibly deforming and updated). For nodes that do not overlap, we can skip the updating of their subtrees, and so save much effort. On the other hand, for nodes that do overlap, a top-down strategy is used for updating nodes in their subtrees as they are needed during a tree traversal. Bottom-up could also be used for these, but top-down was found to be more effective [56]. Good results were achieved when the first $n/2$ upper levels were updated with a bottom-up method, and the lower $n/2$ levels with a top-down method. This is shown in Figure 25.20. To update a node top-down, the node records which vertices its entire subtree holds, and this list is traversed and a minimal BV is computed. When using top-down update, overlap testing is done as soon as a BV has been updated, so that the traversal can be terminated if no overlap is found. This, again, gives more efficiency. Initial tests show that this method is four to five times faster than van den Bergen's method [56].

Sometimes, one can create more efficient algorithms when there is some known information about the type of deformation. For example, if a model is deformed using morphing (Section 4.5), then you can morph (i.e., blend) the BVs in a bounding

volume hierarchy in the same way that you morph the actual geometry [57]. This does not create optimal BVs during morphing, but they are guaranteed to always contain the morphed geometry. This update can also be done in a top-down manner, i.e., only where it is needed. This technique can be used for AABBs, k-DOPs, and spheres. Computing a morphed BV from k different BVs costs $O(k)$, but usually k is minuscule and can be regarded as a constant. James and Pai [43] present a framework with a reduced deformation model, which is described by a combination of displacement fields. This provides for generous increases in performance. Ladislav and Zara present similar CD techniques for skinned models [52].

However, when the motion is completely unstructured and has breaking objects, these methods do not work. Some recent techniques attempt to track how well a subtree in a BVH fits the underlying geometry, and only rebuilds them as needed using some heuristics [58, 94].

Another general method for deformable objects first computes minimal AABBs around two objects to be tested for collision [83]. If they overlap, the overlap AABB region is computed, which is simply the intersection volume of the AABBs. It is only inside this overlap region that a collision can occur. A list of all triangles inside this region is created. An octree (Section 19.1.3) that surrounds the entire scene is then used. The idea is then to insert triangles into the octree's nodes, and if triangles from both objects are found in a leaf node, then these triangles are tested against each other. Several optimizations are possible. First, the octree does not need to be built explicitly. When a node gets a list of triangles, the triangles are tested against the eight children nodes, and eight new triangle lists are created. This recursion ends at the leaves, where triangle/triangle testing can take place. Second, this recursion can end any time the triangle list has triangles from only one of the objects. To avoid testing a triangle pair more than once, a checklist is kept that tracks the tested pairs. The efficiency of this method may break down when an overlap region is extensive, or there are many triangles in an overlap region.

Another approach is to use the GPU tessellation capability to detect collisions [72]. First the region of overlap between two objects are found, and then all patches inside that region are voxelized using tessellation using one bit per voxel. This voxel grid is then used to detect collisions. This approach was extended to also handle displacement mapping due to, for example, a car driving in sand and leaving trails [79].

25.8 Continuous Collision Detection

Often, collision detection is simply done once per frame and at discrete times. Dynamic intersection tests were presented in Section 25.11, where the time of impact between simple objects were computed. The reason for doing such tests is to avoid strange artifacts. For example, if a ball is being thrown in high speed toward a wall, the ball may be in front of the wall at one time, and for the next frame it has moved past the wall and escaped the detection of collision with the wall. Continuous collision detection (CCD), similar to dynamic intersection tests, aims at removing such artifacts.

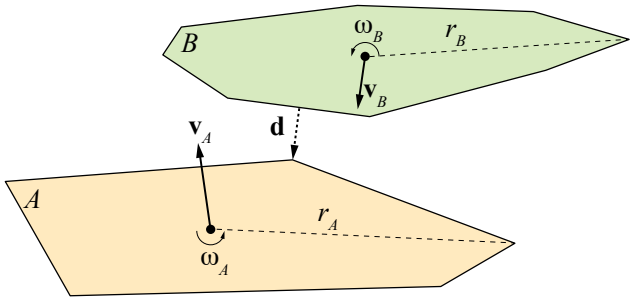

Figure 25.21. Notation for continuous collision detection used in Equation 25.11. (Illustration after Catto [14].)

Here, we will describe a method that is used in the game industry. We wish to determine when two convex objects A and B collide. Each object has a linear velocity $\mathbf{v}$ and an angular velocity ω, and a radius r which is the distance from the centroid of the object to the farthest vertex from it. This notation is illustrated in Figure 25.21. First, the shortest distance vector, $\mathbf{d}$, between A and B is computed using the GJK algorithm (Section 25.3.2). The idea is then to compute a time Δt where it can be guaranteed that the two objects do not collide inside that time interval. It is possible to prove [14, 97]

$$\underbrace{\left((\mathbf{v}_B - \mathbf{v}_A) \cdot \frac{\mathbf{d}}{||\mathbf{d}||} + ||\omega_A||r_A + ||\omega_B||r_b\right)}_{c} \Delta t \leq ||\mathbf{d}||, \tag{25.11}$$

where c is a bound on the velocity along $\mathbf{d}$. If none of A and B were rotating, $c = (\mathbf{v}_B - \mathbf{v}_A) \cdot \mathbf{d}/||\mathbf{d}||$, i.e., the relative velocity projected on $\mathbf{d}$, which means that the equation is the classic formula where distance equals velocity multiplied by time. The rotational terms (using ω) conservatively estimate the maximum change due to rotation. Hence, the maximum amount of time without a collision is then $\Delta t = ||\mathbf{d}||/c$. If this time is larger than then time between two frames, then A and B cannot collide. If this is not true, then it is possible to move the two objects as far as they can get during Δt, and then run the algorithm again until it converges to some tolerance. Since the estimate Δt is conservative, this algorithm is called conservative advancement. Catto improves performance using root-finding techniques and uses this technique in *Diablo 3* [14]. Zhang et al. [97] describes a method to handle also non-convex polyhedra.

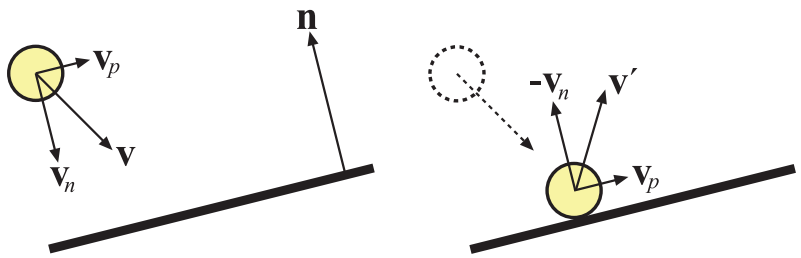

Figure 25.22. Collision response for a sphere approaching a plane. To the left the velocity vector $\mathbf{v}$ is divided into two components, $\mathbf{v}_n$, and $\mathbf{v}_p$. To the right, perfectly elastic (bouncy) collision is shown, where the new velocity is $\mathbf{v}' = \mathbf{v}_p - \mathbf{v}_n$. For a less elastic collision, the length of $-\mathbf{v}_n$ could be decreased.

25.9 Collision Response

Collision response is the action that should be taken to avoid (abnormal) interpenetration of objects. Assume, for example, that a sphere is moving toward a cube. When the sphere first hits the cube, which is determined by collision detection algorithms, we would like the sphere to change its trajectory (e.g., velocity direction), so it appears that they collided. This is the task of *collision response* techniques, which has been and still is the subject of intensive research [2, 18, 38, 69, 71, 93]. It is a complex topic, and in this section only the simplest technique will be presented.

In Section 25.11.1, a technique for computing the exact time of collision between a sphere and a plane was presented. Here we will explain what happens to the sphere's motion at the time of collision.

Assume that a sphere is moving toward a plane. The velocity vector is $\mathbf{v}$, and the plane is $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$, where $\mathbf{n}$ is normalized. This is shown in Figure 25.22. To compute the simplest response, we represent the velocity vector as

$$\mathbf{v} = \mathbf{v}_n + \mathbf{v}_p, \text{ where } \quad \mathbf{v}_n = (\mathbf{v} \cdot \mathbf{n})\mathbf{n}, \quad \text{and } \quad \mathbf{v}_p = \mathbf{v} - \mathbf{v}_n. \tag{25.12}$$

With this representation, the velocity vector, $\mathbf{v}'$, after the collision is [54]

$$\mathbf{v}' = \mathbf{v}_p - \mathbf{v}_n. \tag{25.13}$$

Here, we have assumed that the response was totally elastic. This means that no kinetic energy is lost, and thus that the response is "perfectly bouncy" [93]. Now, normally the ball is deformed slightly at collision, and some energy transformed into heat, so some energy is lost. This is described with a coefficient of *restitution*, k (often also denoted ϵ). The velocity parallel to the plane, $\mathbf{v}_p$, remains unchanged, while $\mathbf{v}_n$ is dampened with $k \in [0, 1]$:

$$\mathbf{v}' = \mathbf{v}_p - k\mathbf{v}_n, \tag{25.14}$$

which is an empirical "law" for collisions. As k gets smaller, more and more energy is lost, and the collision appears less and less bouncy. At $k = 0$, the motion after collision is parallel to the plane, so the ball appears to roll on the plane.

More sophisticated collision response is based on simulation of physics, and involves creating a system of equations that is solved using an *ordinary differential equation* (ODE) solver. In such algorithms, a point of collision and a normal at this point is needed. The interested reader should consult the SIGGRAPH course notes by Witkin et al. [93] and the paper by Dingliana et al. [18]. Catto [15] describes how sequential impulses can be applied using an iterative solver. Also, O'Sullivan and Dingliana present experiments that show that it is hard for a human to judge whether a collision response is correct [74, 75]. This is especially true when more dimensions are involved (i.e., it is easier in one dimension than in three). To produce a real-time algorithm, they found that when there was not time enough to compute an accurate response, a random collision response could be used. These were found to be as believable as more accurate responses.

25.10 Particles

In this section, we will describe two collision detection methods for particles. The first type is for particle systems (Section 13.8), which often are used as special effects, and the other type is more for physics simulations, e.g., to model fluids and smoke, where matter is approximated by a large set of small particles. These two types are closely related and the techniques can sometimes be used for both.

25.10.1 Particle Systems

We start by describing an inexpensive, approximate collision system for particles based on the depth buffer [84]. This is in the same spirit as screen-space ambient occlusion methods (Section 11.3). The basic idea is to provide a method that can handle tens of thousands particles in a tiny amount of total time. The system can be used for, e.g., rain drops, sparks, rock chips, and splashes. Each particle is to be tested for collision against the scene and the approximation used here is the depth buffer as rendered from the eye. This means that a particle can only collide against a visible surface on screen. To avoid particles from moving through a visible surface, it is assumed that each surface as some thickness in depth in which collision occurs. When a collision occurs, the normal is fetched from a normal buffer, and for incoming particles reaching the surface, the particle is first moved back toward its velocity vector to the position in the depth buffer. After that, the reflected direction is computed and the particle is moved in that direction. Particles coming from behind are simply destroyed since doing so avoids having them "bleed through" visible surfaces.

Using the depth buffer has some disadvantages, e.g, that collision only can occur against surfaces visible from the eye, but this method can work admirably for approx-

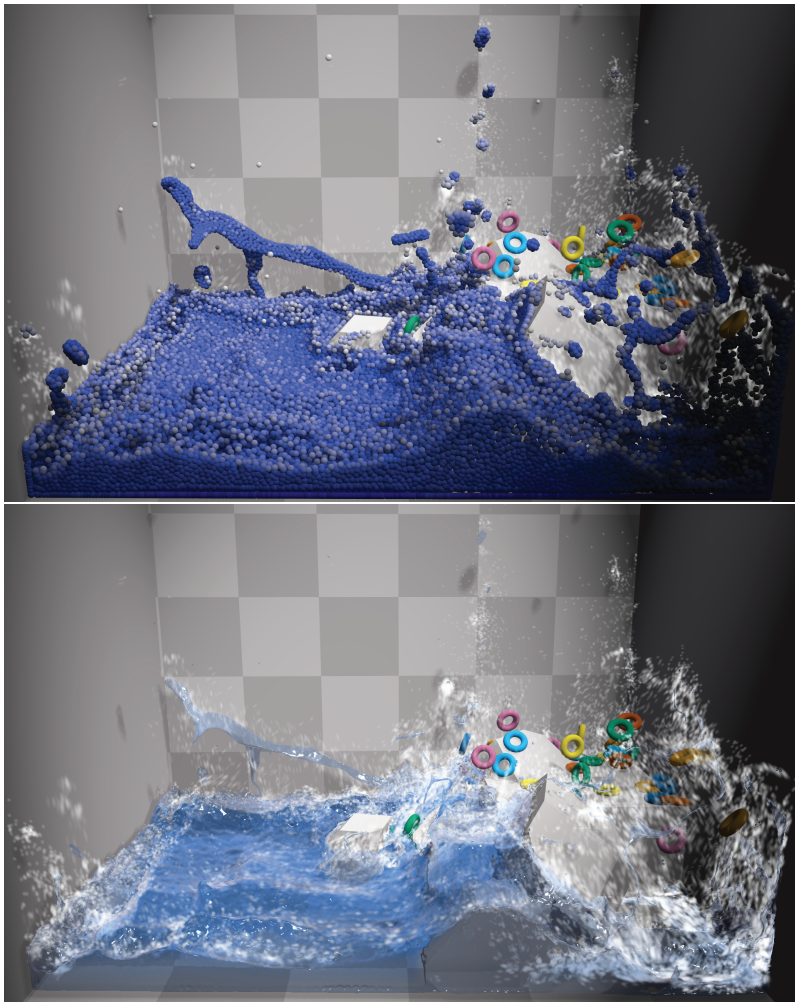

Figure 25.23. Top: A water surface simulated using particles, where each particle is visualized as a blue sphere. The water interacts with both the environment consisting of a few rocks, some tori, walls and floor. Bottom: The water surface derived from the particles was rendered with motion blur and refraction. (*Images simulated and rendered using NVIDIA Flex demo.*)

imate effects. If better accuracy is needed one can use signed distance fields (SDFs), which also are treated in Section 17.3, for collision detection. The major advantage is that the SDF can also be created for hidden parts of the scene and so the representation is often stored in a three-dimensional texture. Fisher and Lin [27] used SDFs to compute penetration distance between deformable models. Furhmann et al. [29]

represented rigid-body objects using only SDFs and then tested only the particles of deformable objects against the SDF. Particles penetrating the inside of the SDF were moved back to the surface itself. More complex methods for both collision detection and response can also be used. Di Donato [17] uses ASTC compressed volume textures on mobile to represent a voxelization of the scene for particle collision detection, along with transform feedback to store results for the next step of the simulation.

25.10.2 Particles for Physical Simulation

Here, we will briefly describe how particles can be used for physical simulation of, say, a volume of water, as illustrated in Figure 25.23. It is common to represent such matter as a large set of particles, where each particle can have any position as long as it respects the other particles, e.g., two particles are not allowed to get too close to each other. Other quantities that particles need to take into account are gravity, velocity, and user-supplied forces, for example. This is called a Lagrangian or particle-based method [35]. Since the force of a particle drops off with the distance, it is therefore sufficient to let a particle interact with only the particles within a certain radius. Hence, we can use any type of spatial data structure to accelerate the process of finding particle-particle collisions. This includes quickly building an AABB tree (Section 25.1.3) or uniform or hierarchical grids (Section 25.1.3), for example.

Macklin et al. [64] present an approach for handling liquids, gases, rigid bodies, cloth, and deformable solids in a single unified framework so that all types of matter can affect all other types. They use signed distance fields to rapidly resolve collisions. They present a GPU implementation with real-time performance. An example is shown in Figure 25.23.

25.11 Dynamic Intersection Testing

In Chapter 22, only *static* intersection testing is considered. This means that all objects involved are not moving when tested. However, this is not always a realistic scenario, especially since we render frames at discrete times. For example, discrete testing means that a ball that is on one side of a closed door at time t might move to the other side at $t + \Delta t$ (i.e., the next frame), without any collision being noticed by a static intersection test. One solution is to make several tests uniformly spaced between t and $t + \Delta t$. This would increase the computational load, and still the intersection could be missed. The thickness of the far side of the door could be increased for testing purposes, but if the sphere is moving quickly enough this technique may fail [14]. A *dynamic* intersection test is designed to cope with this problem. Section 25.5 provided an in-depth examination of dynamic intersection testing of a BSP with a cylinder. This section provides other common dynamic intersection tests. More information can be found in Ericson's [25] and Eberly's [22] books, as well as presentations by Catto [14] and Gregorius [37].

Methods such as shaft culling [39] can be used to aid in intersection testing of moving AABBs. The object moving through space is represented by two AABBs at different times, and these two AABBs are joined by a small set of planes. This simple convex hull can be tested against objects for intersection. However, bounding sphere intersection algorithms are considerably faster to evaluate and can be more efficient overall if the spheres contain their objects fairly tightly. In fact, it is often worthwhile to use a small set of spheres to tightly bound and represent the moving object [81]. Capsules—two spheres joined by a tube—can be used for the bounds of animated characters, and tapered capsules for limbs and cloth simulation.

One principle that can be applied to dynamic intersection testing situations where only translations (not rotations) take place is the fact that motion is relative. Assume object A moves with velocity $\mathbf{v}_A$ and object B with velocity $\mathbf{v}_B$, where the velocity is the amount an object has moved during the frame. To simplify calculations, we instead assume that A is moving and B is still. To compensate for B's velocity, A's velocity is then: $\mathbf{v} = \mathbf{v}_A - \mathbf{v}_B$. As such, only one object is given a velocity in the algorithms that follow.

25.11.1 Sphere/Plane

Testing a sphere dynamically against a plane is simple. Assume the sphere has its center at $\mathbf{c}$ and a radius r. In contrast to the static test, the sphere also has a velocity $\mathbf{v}$ during the entire frame time Δt. So, in the next frame, the sphere will be located at $\mathbf{e} = \mathbf{c} + \Delta t\mathbf{v}$. For simplicity, assume Δt is 1 and that this frame starts at time 0. The question is: Has the sphere collided with a plane $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ during this time?

The signed distance, s_c, from the sphere's center to the plane is obtained by plugging the sphere center into the plane equation. Subtracting the sphere radius from this distance gives how far (along the plane normal) the sphere can move before reaching the plane. This is illustrated in Figure 25.24. A similar distance, s_e, is computed for the endpoint $\mathbf{e}$. Now, if the sphere centers are on the same side of the plane (tested as $s_c s_e > 0$), and if $|s_c| > r$ and $|s_e| > r$, then an intersection cannot occur, and the sphere can safely be moved to $\mathbf{e}$. Otherwise, the sphere position and the exact time when the intersection occurs is obtained as follows [32]. The time when the sphere first touches the plane is t, where t is computed as

$$t = \frac{s_c - r}{s_c - s_e}. \tag{25.15}$$

The sphere center is then located at $\mathbf{c} + t\mathbf{v}$. A simple collision response at this point would be to reflect the velocity vector $\mathbf{v}$ around the plane normal, and move the sphere using this vector: $(1 - t)\mathbf{r}$, where $1 - t$ is the remaining time to the next frame from the collision, and $\mathbf{r}$ is the reflection vector.

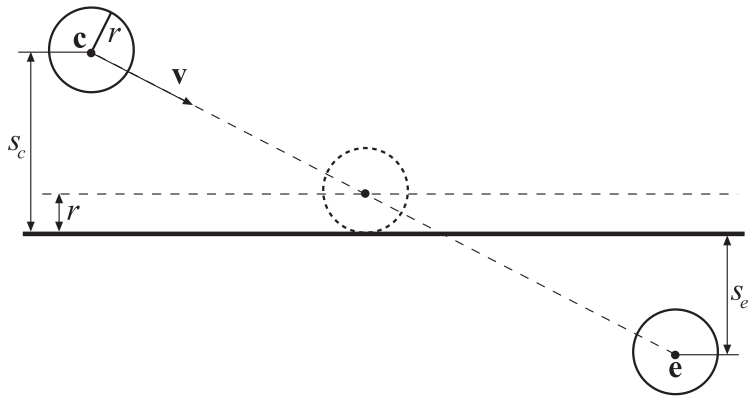

Figure 25.24. The notation used in the dynamic sphere/plane intersection test. The middle sphere shows the position of the sphere at the time when collision occurs. Note that s_c and s_e are both signed distances.

25.11.2 Sphere/Sphere

Testing two moving spheres A and B for intersection turns out to be equivalent to testing a ray against a static sphere—a surprising result. This equivalency is shown by performing two steps. First, use the principle of relative motion to make sphere B become static. Then, a technique is borrowed from the frustum/sphere intersection test (Section 22.14.2). In that test, the sphere was moved along the surface of the frustum to create a larger frustum. By extending the frustum outward by the radius of the sphere, the sphere itself could be shrunk to a point. Here, moving one sphere over the surface of another sphere results in a new sphere that is the sum of the radii of the two original spheres.

So, the radius of sphere A is added to the radius of B to give B a new radius. Now we have the situation where sphere B is static and is larger, and sphere A is a point moving along a straight line, i.e., a ray. See Figure 25.25.

As this basic intersection test was already presented in Section 22.6, we will simply present the final result:

$$(\mathbf{v}_{AB} \cdot \mathbf{v}_{AB})t^2 + 2(\mathbf{l} \cdot \mathbf{v}_{AB})t + \mathbf{l} \cdot \mathbf{l} - (r_A + r_B)^2 = 0. \tag{25.16}$$

In this equation, $\mathbf{v}_{AB} = \mathbf{v}_A - \mathbf{v}_B$, and $\mathbf{l} = \mathbf{c}_A - \mathbf{c}_B$, where $\mathbf{c}_A$ and $\mathbf{c}_B$ are the centers of the spheres. This gives a, b, and c:

$$\begin{aligned} a &= (\mathbf{v}_{AB} \cdot \mathbf{v}_{AB}), \\ b &= 2(\mathbf{l} \cdot \mathbf{v}_{AB}), \\ c &= \mathbf{l} \cdot \mathbf{l} - (r_A + r_B)^2, \end{aligned} \tag{25.17}$$

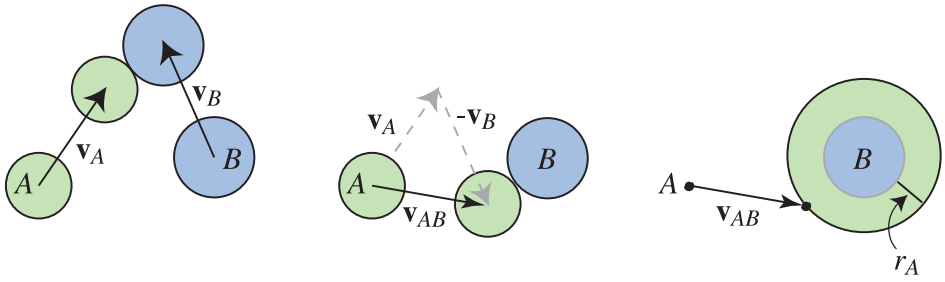

Figure 25.25. The left figure shows two spheres moving and colliding. In the center figure, sphere B has been made static by subtracting its velocity from both spheres. Note that the relative positions of the spheres at the collision point remains the same. On the right, the radius r_A of sphere A is added to B and subtracted from itself, making the moving sphere A into a ray.

which are values used in the quadratic equation

$$at^2 + bt + c = 0. \tag{25.18}$$

The two roots are computed by first computing

$$q = -\frac{1}{2}(b + \text{sign}(b)\sqrt{b^2 - 4ac}). \tag{25.19}$$

Here, sign(b) is $+1$ when $b \geq 0$, else -1. Then the two roots are

$$\begin{aligned} t_0 &= \frac{q}{a}, \\ t_1 &= \frac{c}{q}. \end{aligned} \tag{25.20}$$

This form of solving the quadratic is not what is normally presented in textbooks, but Press et al. note that it is more numerically stable [77].

We assume that the spheres do not overlap at the start, something that could be determined by a static sphere/sphere test. The smallest value in the range $[t_0, t_1]$ that lies within $[0, 1]$ (the time of the frame) is the time of first intersection. Plugging this t-value into

$$\begin{aligned} \mathbf{p}_A(t) &= \mathbf{c}_A + t\mathbf{v}_A, \\ \mathbf{p}_B(t) &= \mathbf{c}_B + t\mathbf{v}_B \end{aligned} \tag{25.21}$$

yields the location of each sphere at the time of first contact. The main difference of this test and the ray/sphere test presented earlier is that the ray direction $\mathbf{v}_{AB}$ is not normalized here.

25.11.3 Sphere/Polygon

Dynamic sphere/plane intersection was simple enough to visualize directly. That said, sphere/plane intersection can be converted to another form in a similar fashion as done with sphere/sphere intersection. That is, the moving sphere can be shrunk to a moving point, so forming a ray, and the plane expanded to a slab the thickness of the sphere's diameter. The key idea used in both tests is that of computing what is called the *Minkowski sum* of the two objects. The Minkowski sum of a sphere and a sphere is a larger sphere equal to the radius of both.

The sum of a sphere and a plane is a plane thickened in each direction by the sphere's radius. Any two volumes can be added together in this fashion, though sometimes the result is difficult to describe. For dynamic sphere/polygon testing the idea is to test a ray against the Minkowski sum of the sphere and polygon.

We will not present his method in depth here, but rather note that this sphere/polygon test is equivalent to testing a ray (represented by the center of the sphere moving along a line) against the Minkowski sum of the sphere and polygon. This summed surface is one in which the vertices have been turned into spheres of radius r, the edges into cylinders of radius r, and the polygon itself duplicated, then raised and lowered by r to seal off the object. See Figure 25.26 for a visualization of this. This is the same sort of expansion as done for frustum/sphere intersection (Section 22.14.2). So, the algorithm presented can be thought of as testing a ray against this volume's parts: First, the polygon facing the ray is tested, then the cylinders representing the edges are tested, and finally, the vertex spheres are tested against the ray.

Thinking about this puffy object gives insight as to why a polygon is most efficiently tested by using the order of area, then edges, then vertices. The polygon in this puffy object that faces the sphere is not covered by the object's cylinders and spheres, so testing it first will give the closest possible intersection point without further testing. Similarly, the cylinders formed by the edges cover the spheres, but the spheres cover only the insides of the cylinders.

Hitting the inside of the cylinder with a ray is equivalent to finding the point where the moving sphere last hits the corresponding edge, a point we do not care about. The closest cylinder exterior intersection (if one exists) will always be closer than the closest sphere intersection. So, finding a closest intersection with a cylinder is sufficient to end testing without needing to check the vertex spheres. It is much easier (at least for us) to think about testing order when dealing with a ray and this puffy object than the original sphere and polygon.

Another insight from this puffy object model is that, for polygons with concavities, a vertex at any concave location does not have to be tested against the moving sphere, as the sphere formed at such a vertex is not visible from the outside. Efficient dynamic sphere/object intersection tests can be derived by using relative motion and the transformation of a moving sphere into a ray by using Minkowski sums.

Static intersection tests can also be derived by testing against the Minkowski sum.

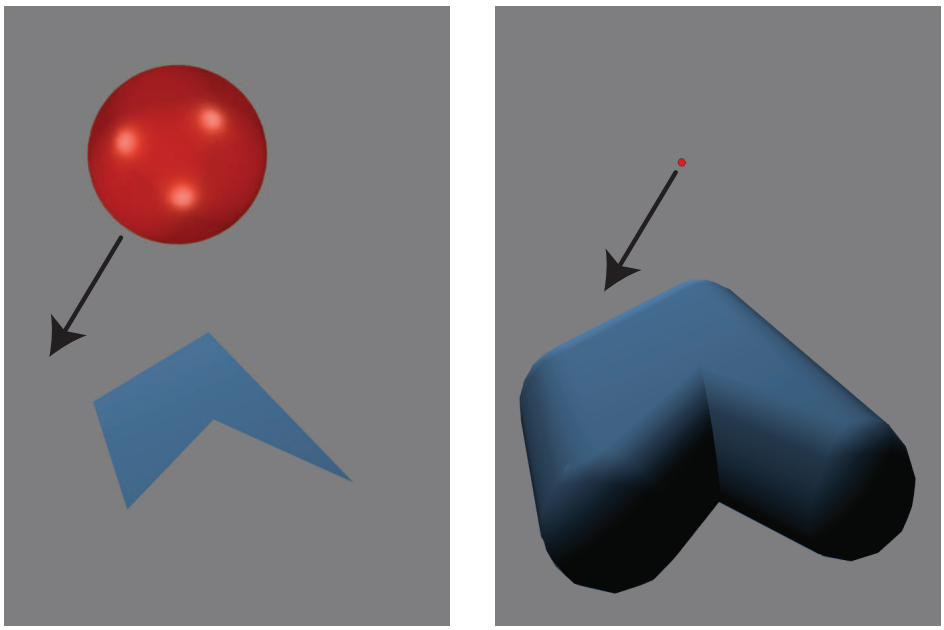

Figure 25.26. In the left figure, a sphere moves toward a polygon. In the right figure, the sphere is deflated to a point and a ray shoots at an "inflated" version of the polygon. The two intersection tests are equivalent.

For example, if the sphere's center is found to be inside this puffy model, the sphere and triangle intersect at their initial locations. The related idea of a Minkowski difference is a more general way to describe this inclusion test for arbitrary objects (Section 25.3.2). Taking the difference subtracts one object from another. The subtracted object has its coordinate values negated. If two objects intersect, the origin will be included in their Minkowski difference. This point represents a location that exists inside both objects. Gregorius [36] discusses how the Minkowski difference and an object's Gauss map can provide insights for optimizing the use of the separating axis test.

More complex dynamic tests exist, such as for triangle/triangle contact. Van Waveren [91] discusses dynamic collision detection for polyhedra, based on testing for vertex/polygon and edge/edge collisions. Catto [14] discusses the concept of *conservative advancement*, where the safe distance between two objects is found and used to move the objects closer and then to test again. However, using this safe interval can make convergence take a long time. She presents bilateral advancement for triangle/triangle intersection, which is effectively an improved root-finding method. Shellshear and Ytterlid [82] provide SSE-optimized code for distance queries between triangles, lines, and points. The *Proximity Query Package* (PQP) library provides code for mesh object overlap and the minimum distance between two models.

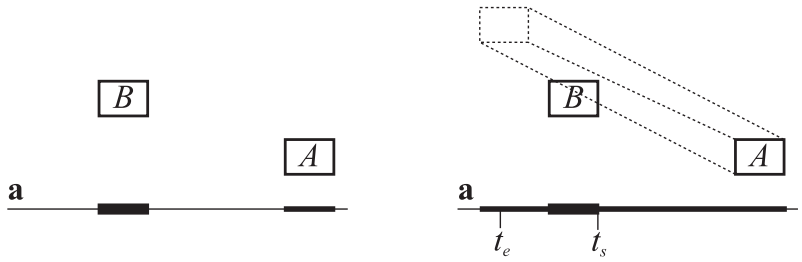

Figure 25.27. Left: the stationary SAT illustrated for an axis **a**. A and B do not overlap on this axis. Right: the dynamic SAT illustrated. A moves and the projection of its interval on **a** is tracked during the movement. Here, the two objects overlap on axis **a**.

25.11.4 Dynamic Separating Axis Method

The separating axis test (SAT) in Section 22.2 is helpful in testing convex polyhedrons, e.g., boxes and triangles, against each other. This type of testing can be extended to dynamic queries as well [11, 20, 25, 37, 80].

Remember that the SAT method tests a set of axes to see whether the projections of the two objects onto these axes overlap. If all projections on all axes overlap, then the objects overlap as well. The key to solving the problem dynamically is to move the projected interval of the moving object with a speed of $(\mathbf{v}\cdot\mathbf{a})/(\mathbf{a}\cdot\mathbf{a})$ (see Equation 4.62) on the axis, **a** [11]. Again, if there is overlap on all tested axes, then the dynamic objects overlap, otherwise they do not. See Figure 25.27 for an illustration of the difference between the stationary SAT and the dynamic SAT.

Eberly [20], using an idea by Ron Levine, also computes the actual time of intersection between A and B. This is done by computing times when they just start to overlap, t_s, and when they stop overlapping (because the intervals have moved "through" each other), t_e. The hit between A and B occurs at the largest of all the t_ss for all the axes. Likewise, the end of overlapping occurs at the smallest of all the t_e values. Early rejection optimizations include detecting when the intervals are nonoverlapping at $t = 0$ and moving apart. Also, if at any time the largest t_s is greater than the smallest t_e, then the objects do not overlap, and so the test is terminated. This is similar to the ray/box intersection test in Section 22.7.1. Eberly has code for a wide range of tests between convex polyhedra, including box/box, triangle/box, and triangle/triangle. Gregorius [37] presents algorithms for dynamic intersection tests among spheres, capsules, convex hulls, and meshes.

Further Reading and Resources

See this book's website, realtimerendering.com, for the latest information and free software in this field. One of the best resources for CD is Ericson's book *Real-Time Collision Detection* [25], which also includes much code. The collision detection book by van den Bergen [9] has a particular focus on GJK and the SOLID CD software

system, included with the book. Van den Bergen has a worthwhile presentation about physics for game programmers [10] and Catto provides a pleasant introduction to GJK [13]. Teschner et al. [87] provide a survey of CD algorithms for deformable objects. Schneider and Eberly present algorithms for computing the distance between many different primitives in their book on geometrical tools [80].

More information on spatial data structures can be found in Section 19.1. Beyond Ericson's more approachable book [25], Samet's book [78] is an extremely comprehensive reference work on spatial data structures.

Books by Millington [68], Erleben et al. [26], and Eberly [21] are comprehensive guides to the field of collision response.

Bibliography

[1] Apetrei, Ciprian, "Fast and Simple Agglomerative LBVH Construction," *Computer Graphics and Visual Computing*, 2014. Cited on p. 12

[2] Baraff, D., "Curved Surfaces and Coherence for Non-Penetrating Rigid Body Simulation," *Computer Graphics (SIGGRAPH '90 Proceedings)*, vol. 24, no. 4, pp. 19–28, Aug. 1990. Cited on p. 33

[3] Baraff, D., *Dynamic Simulation of Non-Penetrating Rigid Bodies*, PhD thesis, Technical Report 92-1275, Computer Science Department, Cornell University, 1992. Cited on p. 4

[4] Barzel, Ronen, ed., *Graphics Tools—The jgt Editors' Choice*, A K Peters, Ltd., 2005. Cited on p. 45, 47

[5] van den Bergen, G., "Efficient Collision Detection of Complex Deformable Models Using AABB Trees," *journal of graphics tools*, vol. 2, no. 4, pp. 1–13, 1997. Also collected in [4]. Cited on p. 14, 15, 29, 30

[6] van den Bergen, G., "A Fast and Robust GJK Implementation for Collision Detection of Convex Objects," *journal of graphics tools*, vol. 4, no. 2, pp. 7–25, 1999. Cited on p. 15, 19, 20

[7] van den Bergen, Gino, *Collision Detection in Interactive 3D Computer Animation*, PhD thesis, Eindhoven University of Technology, 1999. Cited on p. 15, 20, 29

[8] van den Bergen, Gino, "Proximity Queries and Penetration Depth Computation on 3D Game Objects," *Game Developers Conference*, pp. 821–837, Mar. 2001. Cited on p. 20

[9] van den Bergen, Gino, *Collision Detection in Interactive 3D Environments*, Morgan Kaufmann, 2003. Cited on p. 15, 20, 42

[10] van den Bergen, Gino, "Physics for Game Programmers," *Game Developers Conference*, Mar. 2012. Cited on p. 43

[11] Bobic, Nick, "Advanced Collision Detection Techniques," *Gamasutra*, Mar. 2000. Cited on p. 42

[12] Cameron, S., "Enhancing GJK: Computing Minimum and Penetration Distance between Convex Polyhedra," in *Proceedings of International Conference on Robotics and Automation*, IEEE Computer Society, pp. 3112–3117, 1997. Cited on p. 20

[13] Catto, Erin, "Computing Distance," *Game Developers Conference*, Mar. 2010. Cited on p. 43

[14] Catto, Erin, "Continuous Collision," *Game Developers Conference*, Mar. 2013. Cited on p. 32, 36, 41

[15] Catto, Erin, "Understanding Constraints," *Game Developers Conference*, Mar. 2014. Cited on p. 34

[16] Cohen, Jonathan D., Ming C. Lin, Dinesh Manocha, and Madhave Ponamgi, "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scaled Environments," *Symposium on Interactive 3D Graphics*, pp. 189–196, 1995. Cited on p. 3

[17] Di Donato, Daniele, "Implementing a GPU-Only Particles Collision System with ASTC 3D Textures and OpenGL ES 3.0," in Wolfgang Engel, ed., *GPU Pro*^{6}, CRC Press, pp. 369–385, 2015. Cited on p. 36

[18] Dingliana, John, and Carol O'Sullivan, "Graceful Degradation of Collision Handling in Physically Based Animation," *Computer Graphics Forum*, vol. 19, no. 3, pp. 239–247, 2000. Cited on p. 33, 34

[19] Dingliana, John, and Carol O'Sullivan, "Collisions and Adaptive Level of Detail," *Visual Proceedings (SIGGRAPH 2001)*, p. 156, Aug. 2001. Cited on p. 29

[20] Eberly, David, "Testing for Intersection of Convex Objects: The Method of Separating Axes," Technical Report, Magic Software, 2001. Cited on p. 42

[21] Eberly, David, *Game Physics*, Morgan Kaufmann, 2003. Cited on p. 43

[22] Eberly, David, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Second Edition, Morgan Kaufmann, 2006. Cited on p. 36

[23] Ehmann, Stephen A., and Ming C. Lin, "Accelerated Proximity Queries Between Convex Polyhedra Using Multi-Level Voronoi Marching," in *IEEE/RSJ International Conference on Intelligent Robots and Systems 2000*, IEEE Press, pp. 2101–2106, 2000. Cited on p. 20

[24] Ehmann, Stephen A., and Ming C. Lin, "Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition," *Computer Graphics Forum*, vol. 20, no. 3, pp. 500–510, 2001. Cited on p. 20

[25] Ericson, Christer, *Real-Time Collision Detection*, Morgan Kaufmann, 2005. Cited on p. 6, 7, 36, 42, 43

[26] Erleben, Kenny, Jon Sporring, Knud Henriksen, and Henrik Dohlmann, *Physics Based Animation*, Charles River Media, 2005. Cited on p. 43

[27] Fisher, S., and M. C. Lin, "Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields," in *International Conference on Intelligent Robots and Systems*, IEEE Press, pp. 330–336, 2001. Cited on p. 35

[28] Frye, Stephen, Takahiro Harada, Young J. Kim, and Sung-eui Yoon, "Recent Advances in Real-Time Collision and Proximity Computations for Games and Simulations," *Eurographics Tutorials*, 2012. Cited on p. 22

[29] Fuhrmann, A., G. Sobotka, and C. Gross, "Distance Fields for Rapid Collision Detection in Physically Based Modeling," *GraphiCon*, pp. 58–65, 2003. Cited on p. 35

[30] Gilbert, E., D. Johnson, and S. Keerthi, "A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space," *IEEE Journal of Robotics and Automation*, vol. 4, no. 2, pp. 193–203, Apr. 1988. Cited on p. 19, 20

[31] Goldsmith, Jeffrey, and John Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14–20, May 1987. Cited on p. 9, 10

[32] Gomez, Miguel, "Simple Intersection Tests for Games," *Gamasutra*, Oct. 1999. Cited on p. 37

[33] Gottschalk, S., M. C. Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection," in *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, pp. 171–180, Aug. 1996. Cited on p. 2, 13, 14, 17

[34] Gottschalk, Stefan, *Collision Queries Using Oriented Bounding Boxes*, PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 2000. Cited on p. 12

[35] Green, Simon, "CUDA Particles," Technical Report, NVIDIA, June 2008. Cited on p. 36

[36] Gregorius, Dirk, "The Separating Axis Test between Convex Polyhedra," *Game Developers Conference*, Mar. 2013. Cited on p. 41

[37] Gregorius, Dirk, "Robust Contact Creation for Physics Simulations," *Game Developers Conference*, Mar. 2015. Cited on p. 20, 36, 42

[38] Hahn, James K., "Realistic Animation of Rigid Bodies," *Computer Graphics (SIGGRAPH '88 Proceedings)*, vol. 22, no. 4, pp. 299–308, 1988. Cited on p. 33

[39] Haines, Eric, "A Shaft Culling Tool," *journal of graphics tools*, vol. 5, no. 1, pp. 23–26, 2000. Also collected in [4]. Cited on p. 37

[40] Held, M., J. T. Klosowski, and J. S. B. Mitchell, "Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs," in *Proceedings of the 7th Canadian Conference on Computational Geometry*, ACM pp. 205–210, 1995. Cited on p. 14

[41] Hermann, Everton, François Faure, and Bruno Raffin, "Ray-Traced Collision Detection for Deformable Bodies," *International Conference on Computer Graphics Theory and Applications (GRAPP)*, Jan. 2008. Cited on p. 22

[42] Hubbard, Philip M., "Approximating Polyhedra with Spheres for Time-Critical Collision Detection," *ACM Transactions on Graphics*, vol. 15, no. 3, pp. 179–210, 1996. Cited on p. 14, 24, 28

[43] James, Doug L., and Dinesh K. Pai, "BD-Tree: Output-Sensitive Collision Detection for Reduced Deformable Models," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 393–398, Aug. 2004. Cited on p. 31

[44] Jiménez, P., and Thomas C. Torras, "3D Collision Detection: A Survey," *Computers & Graphics*, vol. 25, pp. 269–285, 2001. Cited on p. 19

[45] Karras, Tero, "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d trees," in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, Eurographics Association, pp. 33–37, June 2012. Cited on p. 12

[46] Karras, Tero, and Timo Aila, "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies," in *Proceedings of the 5th High-Performance Graphics Conference*, ACM, pp. 89–99, July 2013. Cited on p. 12

[47] Klosowski, J. T., M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan, "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998. Cited on p. 9, 14

[48] Knuth, Donald E., *The Art of Computer Programming: Sorting and Searching*, vol. 3, Second Edition, Addison-Wesley, 1998. Cited on p. 5

[49] Konečný, Petr, *Bounding Volumes in Computer Graphics*, MSc thesis, Faculty of Informatics, Masaryk University, Brno, Apr. 1998. Cited on p. 14

[50] Krishnan, S., A. Pattekar, M. C. Lin, and D. Manocha, "Spherical Shell: A Higher Order Bounding Volume for Fast Proximity Queries," in *Proceedings of Third International Workshop on the Algorithmic Foundations of Robotics*, A K Peters, Ltd, pp. 122–136, 1998. Cited on p. 14

[51] Kulpa, R., A.-H. Olivierxs, J. Ondřej, and J. Pettré, Julien, "Imperceptible Relaxation of Collision Avoidance Constraints in Virtual Crowds," *ACM Transactions on Graphics*, vol. 30, no. 6, pp. 138:1–138:10, 2011. Cited on p. 29

[52] Ladislav, Kavan, and Zara Jiri, "Fast Collision Detection for Skeletally Deformable Models," *Computer Graphics Forum*, vol. 24, no. 3, pp. 363–372, 2005. Cited on p. 31

[53] Laine, Samuli, and Tero Karras, "Apex Point Map for Constant-Time Bounding Plane Approximation," in *Eurographics Symposium on Rendering—Experimental Ideas & Implementations*, Eurographics Association, pp. 51–55, 2015. Cited on p. 4

[54] Lander, Jeff, "Collision Response: Bouncy, Trouncy, Fun," *Game Developer*, vol. 6, no. 3, pp. 15–19, Mar. 1999. Cited on p. 33

[55] Larsen, E., S. Gottschalk, M. Lin, and D. Manocha, "Fast Proximity Queries with Swept Sphere Volumes," Technical Report TR99-018, Department of Computer Science, University of North Carolina, 1999. Cited on p. 14, 20

[56] Larsson, Thomas, and Tomas Akenine-Möller, "Collision Detection for Continuously Deforming Bodies," in *Eurographics 2001—Short Presentations*, Eurographics Association, pp. 325–333, Sept. 2001. Cited on p. 29, 30

[57] Larsson, Thomas, and Tomas Akenine-Möller, "Efficient Collision Detection for Models Deformed by Morphing," *The Visual Computer*, vol. 19, no. 2, pp. 164–174, 2003. Cited on p. 31

[58] Larsson, Thomas, and Tomas Akenine-Möller, "A Dynamic Bounding Volume Hierarchy for Generalized Collision Detection," *Computers & Graphics*, vol. 30, no. 3, pp. 451–460, 2006. Cited on p. 31

[59] Lauterbach, C., M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009. Cited on p. 11

[60] Lehericey, François, Valérie Gouranton, and Bruno Arnaldi, "New Iterative Ray-Traced Collision Detection Algorithm for GPU Architectures," in *Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology*, ACM, pp. 215–218, 2013. Cited on p. 22

[61] Lin, M. C., and J. Canny, "A Fast Algorithm for Incremental Distance Computation," in *IEEE International Conference on Robotics and Automation*, IEEE Press, pp. 1008–1014, 1991. Cited on p. 20

[62] Lin, M. C., *Efficient Collision Detection for Animation and Robotics*, PhD thesis, University of California, Berkeley, 1993. Cited on p. 4, 18

[63] Liu, Fuchang, Takahiro Harada, Youngeun Lee, and Young J. Kim, "Real-Time Collision Culling of a Million Bodies on Graphics Processing Units," *ACM Transactions on Graphics*, vol. 29, no. 6, pp. 154:1–154:8, 2010. Cited on p. 6

[64] Macklin, M., M. Müller, N. Chentanez, and T.-Y. Kim, "Unified Particle Physics for Real-Time Applications," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 153:1–153:12, 2014. Cited on p. 36

[65] Melax, Stan, "Dynamic Plane Shifting BSP Traversal," in *Graphics Interface 2000*, Canadian Human-Computer Communications Society, pp. 213–220, May 2000. Cited on p. 23, 24, 26

[66] Melax, Stan, "BSP Collision Detection as Used in *MDK2* and *NeverWinter Nights*," *Gamasutra*, Mar. 2001. Cited on p. 23

[67] Mendoza, Cesar, and Carol O'Sullivan, "Interruptible Collision Detection for Deformable Objects," *Computer and Graphics*, vol. 30, no. 2, pp. 432–438, 2006. Cited on p. 29

[68] Millington, Ian, *Game Physics Engine Development*, Morgan Kaufmann, 2007. Cited on p. 43

[69] Mirtich, Brian, and John Canny, "Impulse-Based Simulation of Rigid-Bodies," in *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, ACM, pp. 181–188, Apr. 1995. Cited on p. 33

[70] Mirtich, Brian, "V-Clip: Fast and Robust Polyhedral Collision Detection," *ACM Transactions on Graphics*, vol. 17, no. 3, pp. 177–208, July 1998. Cited on p. 20

[71] Moore, Matthew, and Jane Wilhelms, "Collision Detection and Response for Computer Animation," *Computer Graphics (SIGGRAPH '88 Proceedings)*, vol. 22, no. 4, pp. 289–298, Aug. 1988. Cited on p. 33

[72] Nießner, M., C. Siegl, H. Schäfer, and C. Loop, "Real-Time Collision Detection for Dynamic Hardware Tessellated Objects," in *Eurographics 2013—Short Papers*, Eurographics Association, pp. 33-36, May 2013. Cited on p. 31

[73] Nunes, Gustavo Bastos, "A 3D Visualization Tool Used for Test Automation in the *Forza* Series," in Wolfgang Engel, ed., *GPU Pro*^{7}, CRC Press, pp. 231–244, 2016. Cited on p. 9

[74] O'Sullivan, Carol, and John Dingliana, "Real vs. Approximate Collisions: When Can We Tell the Difference?," in *ACM SIGGRAPH Sketches and Applications*, ACM, p. 249, Aug. 2001. Cited on p. 28, 34

[75] O'Sullivan, Carol, and John Dingliana, "Collisions and Perception," *ACM Transactions on Graphics*, vol. 20, no. 3, pp. 151–168, 2001. Cited on p. 28, 34

[76] Pouchol, M., A. Ahmad, B. Crespin, and O. Terraz, "A hierarchical hashing scheme for Nearest Neighbor Search and Broad-Phase Collision Detection," *Journal of Graphics, GPU, and Game Tools*, vol. 14, no. 2, pp. 45–59, 2009. Cited on p. 7

[77] Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C*, Cambridge University Press, 1992. Cited on p. 39

[78] Samet, Hanan, *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann, 2006. Cited on p. 43

[79] Schäfer, H., B. Keinert, M. Nießner, C. Buchenau, M. Guthe, and M. Stamminger, "Real-Time Deformation of Subdivision Surfaces from Object Collisions," *High-Performance Graphics*, June 2014. Cited on p. 31

[80] Schneider, Philip, and David Eberly, *Geometric Tools for Computer Graphics*, Morgan Kaufmann, 2003. Cited on p. 42, 43

[81] Schroeder, Tim, "Collision Detection Using Ray Casting," *Game Developer*, vol. 8, no. 8, pp. 50–56, Aug. 2001. Cited on p. 37

[82] Shellshear, Evan, and Robin Ytterlid, "Fast Distance Queries for Triangles, Lines, and Points Using SSE Instructions," *Journal of Computer Graphics Techniques*, vol. 3, no. 4, pp. 86–110, 2014. Cited on p. 41

[83] Smith, Andrew, Yoshifumi Kitamura, Haruo Takemura, and Fumio Kishino, "A Simple and Efficient Method for Accurate Collision Detection Among Deformable Polyhedral Objects in Arbitrary Motion," in *IEEE Virtual Reality Annual International Symposium*, IEEE Computer Society, pp. 136–145, 1995. Cited on p. 31

[84] Tchou, Chris, "Halo Reach Effects Tech," *Game Developers Conference*, Mar. 2011. Cited on p. 34

[85] Terdiman, Pierre, "Faster Convex-Convex SAT: Internal Objects," *Code Corner* blog, Jan. 24, 2011. Cited on p. 20

[86] Teschner, M., B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross, "Optimized Spatial Hashing for Collision Detection of Deformable Objects," in *Proceedings of the Vision, Modeling, and Visualization Conference 2003*, Aka GmbH, pp. 47–54, Nov. 2003. Cited on p. 6

[87] Teschner, M., S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision Detection for Deformable Objects," *Computer Graphics Forum*, vol. 24, no. 1, pp. 61–81, 2005. Cited on p. 43

[88] Turk, Greg, *Interactive Collision Detection for Molecular Graphics*, Technical Report TR90-014, University of North Carolina at Chapel Hill, 1990. Cited on p. 6

[89] Wald, Ingo, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley, "State of the Art in Ray Tracing Animated Scenes," *Computer Graphics Forum*, vol. 28, no. 6, pp. 1691–1722, 2009. Cited on p. 16

[90] Wald, Ingo, Sven Woop, Carsten Benthin, Gregory S. Johnsson, and Manfred Ernst, "Embree: A Kernel Framework for Efficient CPU Ray Tracing," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 143:1–143:8, 2014. Cited on p. 11

[91] van Waveren, J. M. P., "Robust Continuous Collision Detection Between Arbitrary Polyhedra Using Trajectory Parameterization of Polyhedral Features," Technical Report, Id Software, Mar. 2005. Cited on p. 41

[92] Weghorst, H., G. Hooper, and D. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Transactions on Graphics*, vol. 3, no. 1, pp. 52–69, 1984. Cited on p. 13

[93] Witkin, Andrew, David Baraff, and Michael Kass, *SIGGRAPH Physically Based Modeling course*, Aug. 2001. Cited on p. 4, 5, 33, 34

[94] Yoon, Sung-Eui, Sean Curtis, and Dinesh Manocha, "Ray Tracing Dynamic Scenes using Selective Restructuring," in *18th Eurographics Symposium on Rendering*, Eurographics Association, pp. 73–84, June 2007. Cited on p. 31

[95] Ytterlid, R., and E. Shellshear, "BVH Split Strategies for Fast Distance Queries," *Journal of Computer Graphics Techniques*, vol. 4, no. 1, pp. 1–25, 2015. Cited on p. 10, 16

[96] Zachmann, Gabriel, "Rapid Collision Detection by Dynamically Aligned DOP-Trees," in *IEEE Virtual Reality Annual International Symposium*, IEEE Computer Society, pp. 90–97, Mar. 1998. Cited on p. 14

[97] Zhang, X., M. Lee, and Y. J. Kim, "Interactive Continuous Collision Detection for Non-convex Polyhedra," *The Visual Computer*, vol. 22, no. 9, pp. 749–760, 2006. Cited on p. 32

Index