Chapter 4

Bounding Volumes

Directly testing the geometry of two objects for collision against each other is often very expensive, especially when objects consist of hundreds or even thousands of polygons. To minimize this cost, object bounding volumes are usually tested for overlap before the geometry intersection test is performed.

A *bounding volume* (BV) is a single simple volume encapsulating one or more objects of more complex nature. The idea is for the simpler volumes (such as boxes and spheres) to have cheaper overlap tests than the complex objects they bound. Using bounding volumes allows for fast overlap rejection tests because one need only test against the complex bounded geometry when the initial overlap query for the bounding volumes gives a positive result (**Figure 4.1**).

Of course, when the objects really do overlap, this additional test results in an increase in computation time. However, in most situations few objects are typically close enough for their bounding volumes to overlap. Therefore, the use of bounding volumes generally results in a significant performance gain, and the elimination of complex objects from further tests well justifies the small additional cost associated with the bounding volume test.

For some applications, the bounding volume intersection test itself serves as a sufficient proof of collision. Where it does not, it is still generally worthwhile pruning the contained objects so as to limit further tests to the polygons contained in the overlap of the bounding volumes. Testing the polygons of an object A against the polygons of an object B typically has an $O(n^2)$ complexity. Therefore, if the number of polygons to be tested can be, say, cut in half, the workload will be reduced by 75%. **Chapter 6**, on bounding volume hierarchies, provides more detail on how to prune object and polygon testing to a minimum. In this chapter, the discussion is limited to tests of pairs of bounding volumes. Furthermore, the tests presented here are primarily homogeneous in that bounding volumes of the same type are tested against each other. It is not uncommon, however, to use several types of bounding volumes at the same time. Several nonhomogeneous BV intersection tests are discussed in the next chapter.

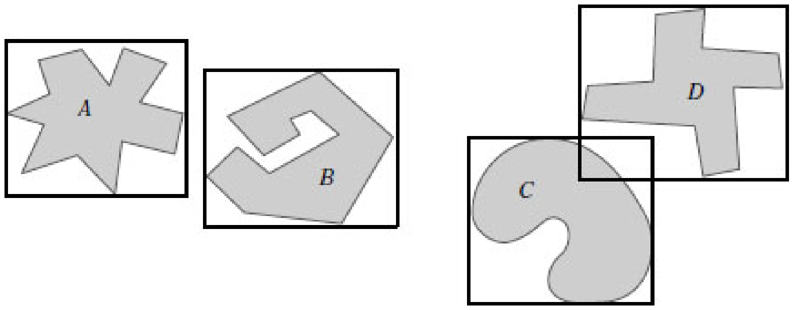

Figure 4.1 The bounding volumes of *A* and *B* do not overlap, and thus *A* and *B* cannot be intersecting. Intersection between *C* and *D* cannot be ruled out because their bounding volumes overlap.

Many geometrical shapes have been suggested as bounding boxes. This chapter concentrates on the shapes most commonly used; namely, spheres, boxes, and convex hull-like volumes. Pointers to a few less common bounding volumes are provided in Section 4.7.

4.1 Desirable BV Characteristics

Not all geometric objects serve as effective bounding volumes. Desirable properties for bounding volumes include:

- Inexpensive intersection tests
- Tight fitting
- Inexpensive to compute
- Easy to rotate and transform
- Use little memory

The key idea behind bounding volumes is to precede expensive geometric tests with less expensive tests that allow the test to exit early, a so-called "early out." To support inexpensive overlap tests, the bounding volume must have a simple geometric shape. At the same time, to make the early-out test as effective as possible the bounding volume should also be as tight fitting as possible, resulting in a trade-off between tightness and intersection test cost. The intersection test does not necessarily just cover comparison against volumes of the same type, but might also test against other types of bounding volumes. Additionally, testing may include queries such as point inclusion, ray intersection with the volume, and intersection with planes and polygons.

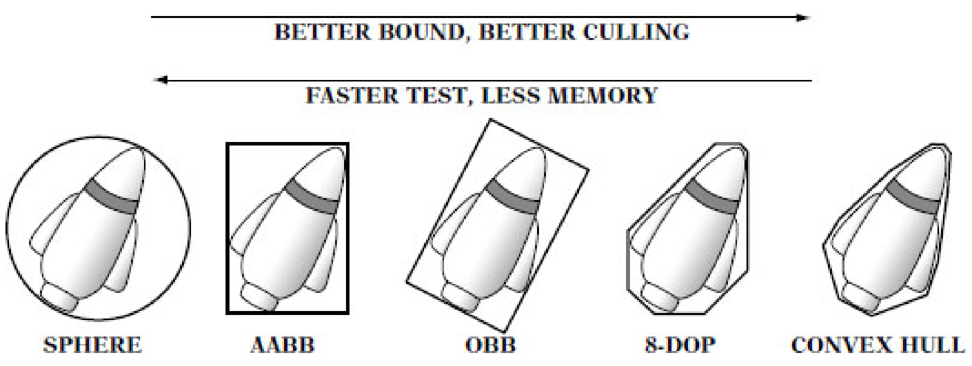

Figure 4.2 Types of bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), eight-direction discrete orientation polytope (8-DOP), and convex hull.

Bounding volumes are typically computed in a preprocessing step rather than at runtime. Even so, it is important that their construction does not negatively affect resource build times. Some bounding volumes, however, must be realigned at runtime when their contained objects move. For these, if the bounding volume is expensive to compute realigning the bounding volume is preferable (cheaper) to recomputing it from scratch.

Because bounding volumes are stored in addition to the geometry, they should ideally add little extra memory to the geometry. Simpler geometric shapes require less memory space. As many of the desired properties are largely mutually exclusive, no specific bounding volume is the best choice for all situations. Instead, the best option is to test a few different bounding volumes to determine the one most appropriate for a given application. Figure 4.2 illustrates some of the trade-offs among five of the most common bounding volume types. The given ordering with respect to better bounds, better culling, faster tests, and less memory should be seen as a rough, rather than an absolute, guide. The first of the bounding volumes covered in this chapter is the axis-aligned bounding box, described in the next section.

4.2 Axis-aligned Bounding Boxes (AABBs)

The *axis-aligned bounding box* (AABB) is one of the most common bounding volumes. It is a rectangular six-sided box (in 3D, four-sided in 2D) categorized by having its faces oriented in such a way that its face normals are at all times parallel with the axes of the given coordinate system. The best feature of the AABB is its fast overlap check, which simply involves direct comparison of individual coordinate values.

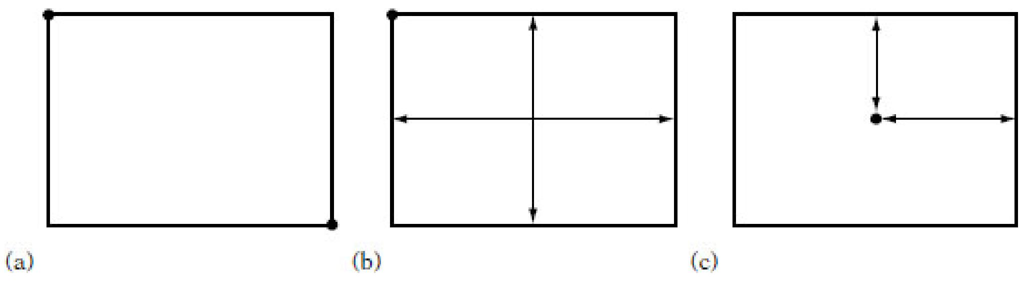

Figure 4.3 The three common AABB representations: (a) min-max, (b) min-widths, and (c) center-radius.

by the minimum and maximum coordinate values along each axis:

```
// region R = {(x, y, z) | min.x<=x<=max.x, min.y<=y<=max.y, min.z<=z<=max.z }
struct AABB {
   Point min;
   Point max;
};
```

This representation specifies the BV region of space as that between the two opposing corner points: *min* and *max*. Another representation is as the minimum corner point *min* and the width or diameter extents *dx*, *dy*, and *dz* from this corner:

```
// region R = {(x, y, z) | min.x<=x<=min.x+dx, min.y<=y<=min.y+dy, min.z<=z<=min.z
struct AABB {
   Point min;
   float d[3];    // diameter or width extents (dx, dy, dz)
};
```

The last representation specifies the AABB as a center point *C* and halfwidth extents or radii *rx, ry*, and *rz* along its axes:

```
// region R = {(x, y, z) | |c.x-x|<=rx,|c.y-y|<=ry, |c.z-z|<=rz }
struct AABB {
   Point c; // center point of AABB
```

```
    float r[3]; // radius or halfwidth extents (rx, ry, rz)
};
```

In terms of storage requirements, the center-radius representation is the most efficient, as the halfwidth values can often be stored in fewer bits than the center position values. The same is true of the width values of the min-width representation, although to a slightly lesser degree. Worst is the min-max representation, in which all six values have to be stored at the same precision. Reducing storage requires representing the AABB using integers, and not floats, as used here. If the object moves by translation only, updating the latter two representations is cheaper than the min-max representation because only three of the six parameters have to be updated. A useful feature of the center-radius representation is that it can be tested as a bounding sphere as well.

4.2.1 AABB-AABB Intersection

Overlap tests between AABBs are straightforward, regardless of representation. Two AABBs only overlap if they overlap on all three axes, where their extent along each dimension is seen as an interval on the corresponding axis. For the min-max representation, this interval overlap test becomes:

```
int TestAABBAABB(AABB a, AABB b)
{
    // Exit with no intersection if separated along an axis
    if (a.max[0] < b.min[0] || a.min[0] > b.max[0]) return 0;
    if (a.max[1] < b.min[1] || a.min[1] > b.max[1]) return 0;
    if (a.max[2] < b.min[2] || a.min[2] > b.max[2]) return 0;
    // Overlapping on all axes means AABBs are intersecting
    return 1;
}
```

The min-width representation is the least appealing. Its overlap test, even when written in an economical way, still does not compare with the first test in terms of number of operations performed:

```
int TestAABBAABB(AABB a, AABB b)
{
    float t;
    if ((t = a.min[0] - b.min[0]) > b.d[0] || -t > a.d[0]) return 0;
    if ((t = a.min[1] - b.min[1]) > b.d[1] || -t > a.d[1]) return 0;
```

```
        if ((t = a.min[2] - b.min[2]) > b.d[2] || -t > a.d[2]) return 0;
        return 1;
    }
```

Finally, the center-radius representation results in the following overlap test:

```
int TestAABBAABB(AABB a, AABB b)
{
   if (Abs(a.c[0] - b.c[0]) > (a.r[0] + b.r[0])) return 0;
   if (Abs(a.c[1] - b.c[1]) > (a.r[1] + b.r[1])) return 0;
   if (Abs(a.c[2] - b.c[2]) > (a.r[2] + b.r[2])) return 0;
   return 1;
}
```

On modern architectures, the **Abs()** call typically translates into just a single instruction. If not, the function can be effectively implemented by simply stripping the sign bit of the binary representation of the floating-point value. When the AABB fields are declared as integers instead of floats, an alternative test for the center-radius representation can be performed as follows. With integers, overlap between two ranges [*A*, *B*] and [*C*, *D*] can be determined by the expression

```
overlap = (unsigned int)(B - C) <= (B - A) + (D - C);
```

By forcing an unsigned underflow in the case when *C* > *B*, the left-hand side becomes an impossibly large value, rendering the expression false. The forced overflow effectively serves to replace the absolute value function call and allows the center-radius representation test to be written as:

```
int TestAABBAABB(AABB a, AABB b)
{
   int r;
   r = a.r[0] + b.r[0]; if ((unsigned int)(a.c[0] - b.c[0] + r) > r + r) return 0;
   r = a.r[1] + b.r[1]; if ((unsigned int)(a.c[1] - b.c[1] + r) > r + r) return 0;
   r = a.r[2] + b.r[2]; if ((unsigned int)(a.c[2] - b.c[2] + r) > r + r) return 0;
   return 1;
}
```

Working in integers allows other implementational tricks, many of which are architecture dependent. SIMD instructions, if present, typically allow

AABB tests to be implemented in just a few instructions worth of code (examples of which are found in **Chapter 13**). Finally, in a collision detection system that has to perform a massive number of overlap tests it may be worthwhile ordering the tests according to the likelihood of their being taken. For instance, if operations largely take place in an almost flat *xz* plane the y-coordinate test should be performed last, as it is least discriminatory.

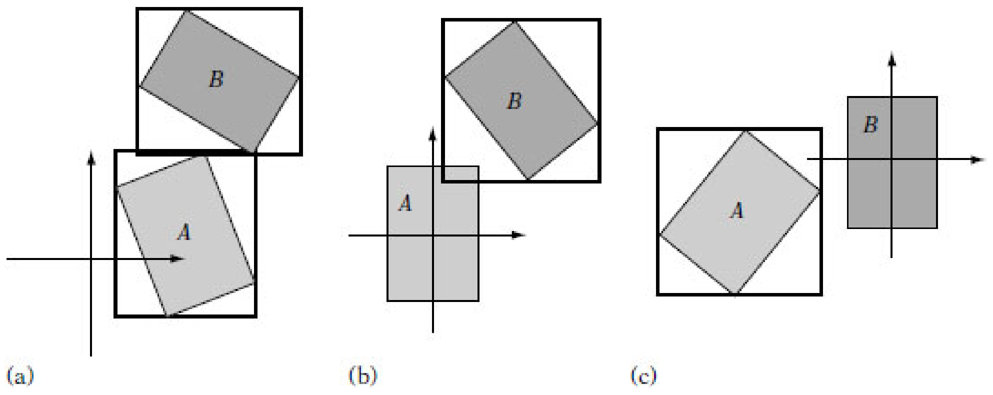

Figure 4.4 (a) AABBs *A* and *B* in world space. (b) The AABBs in the local space of *A*. (c) The AABBs in the local space of B.

4.2.2 Computing and Updating AABBs

Bounding volumes are usually specified in the local model space of the objects they bound (which maybe world space). To perform an overlap query between two bounding volumes, the volumes must be transformed into a common coordinate system. The choice stands between transforming both bounding volumes into world space and transforming one bounding volume into the local space of the other. One benefit of transforming into local space is that it results in having to perform half the work of transformation into world space. It also often results in a tighter bounding volume than does transformation into world space. **Figure 4.4** illustrates the concept. The recalculated AABBs of objects *A* and *B* overlap in world space (**Figure 4.4a**). However, in the space of object B, the objects are found to be separated (**Figure 4.4c**).

Accuracy is another compelling reason for transforming one bounding volume into the local space of the other. A world space test may move both objects far away from the origin. The act of adding in the translation during transformation of the local near-origin coordinates of the bounding volume can force many (or even all) bits of precision of the original values to be lost. For local space tests, the objects are kept near the origin and accuracy is maintained in the calculations. Note, however, that by adjusting the translations so that the transformed objects are centered on the origin world space transformations can be made to maintain accuracy as well.

Transformation into world space becomes interesting when updated bounding volumes can be temporarily cached for the duration of a time step. By caching a bounding volume after transformation, any bounding volume has to be transformed just once into any given space. As all bounding volumes are transformed into the same space when transforming into world space, this becomes a win in situations in which objects are being checked for overlap multiple times. In contrast, caching updated bounding volumes does not help at all when transforming into the local space of other bounding volumes, as all transformations involve either new objects or new target coordinate systems. Caching of updated bounding volumes has the drawback of nearly doubling the required storage space, as most fields of a bounding volume representation are changed during an update.

Some bounding volumes, such as spheres or convex hulls, naturally transform into any coordinate system, as they are not restricted to specific orientations. Consequently, they are called nonaligned or (freely) oriented bounding volumes. In contrast, aligned bounding volumes (such as AABBs) are restricted in what orientations they can assume. The aligned bounding volumes must be realigned as they become unaligned due to object rotation during motion. For updating or reconstructing the AABB, there are four common strategies:

- Utilizing a fixed-size loose AABB that always encloses the object
- Computing a tight dynamic reconstruction from the original point set
- Computing a tight dynamic reconstruction using hill climbing
- Computing an approximate dynamic reconstruction from the rotated AABB

The next four sections cover these approaches in more detail.

4.2.3 AABB from the Object Bounding Sphere

The first method completely circumvents the need to reshape the AABB by making it large enough to contain the object at any orientation. This fixed-size encompassing AABB is computed as the bounding box of the bounding sphere of the contained object A. The bounding sphere, in turn, is centered in the pivot point *P* that *A* rotates about. Its radius *r* is the distance to the farthest object vertex from this center (as illustrated in **Figure 4.5**). By making sure the object pivot *P* lies in the center of the object, the sphere radius is minimized.

The benefit of this representation is that during update this AABB simply need be translated (by the same translation applied to the bounded object), and any object rotation can be completely ignored. However, the bounding sphere itself (which has a better sound than the AABB) would also have this property. Thus, bounding spheres should be considered a potential better choice of bounding volume in this case.

4.2.4 AABB Reconstructed from Original Point Set

The update strategy described in this section (as well as the remaining two update strategies to be described) dynamically resizes the AABB as it is being realigned with the coordinate system axes. For a tightly fitted boundingbox, the underlying geometry of the bounded object is examined and the box bounds are established by finding the extreme vertices in all six directions of the coordinate axes. The straightforward approach loops through all vertices, keeping track of the vertex most distant along the direction vector. This distance can be computed through the projection of the vertex vector onto the direction vector. For comparison reasons, it is not necessary to normalize the direction vector. This procedure is illustrated in the following code, which finds both the least and most distant points along a direction vector:

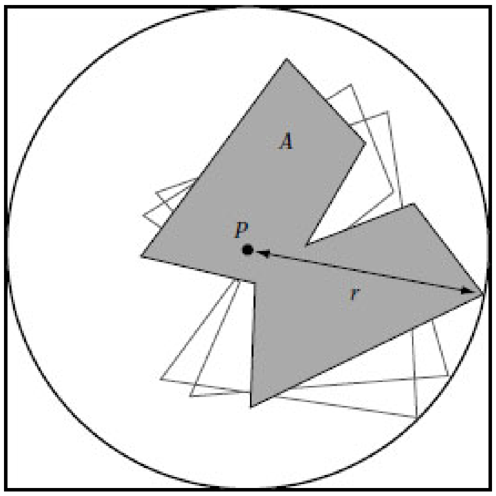

Figure 4.5 AABB of the bounding sphere that fully contains object *A* under an arbitrary orientation.

```
// Returns indices imin and imax into pt[] array of the least and
// most, respectively, distant points along the direction dir
void ExtremePointsAlongDirection(Vector dir, Point pt[], int n, int *imin, int *im
{
   float minproj = FLT_MAX, maxproj = -FLT_MAX;
   for(inti=0;i<n;
     // Project vector from origin to point onto direction vector
     float proj = Dot(pt[i], dir);
     // Keep track of least distant point along direction vector
     if (proj < minproj) {
         minproj = proj;
         *imin = i;
     }
     // Keep track of most distant point along direction vector
     if (proj > maxproj) {
         maxproj = proj;
         *imax = i;
     }
   }
}
```

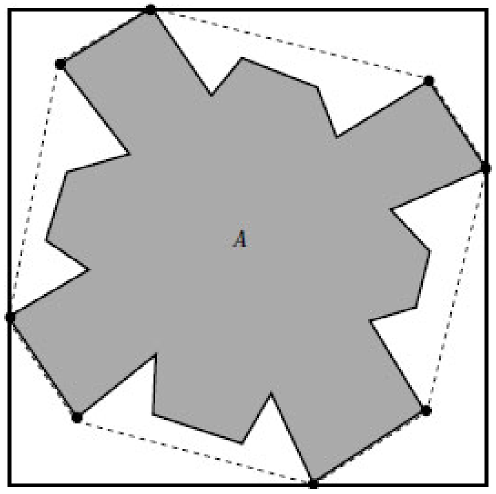

Figure 4.6 When computing a tight AABB, only the highlighted vertices that lie on the convex hull of the object must be considered.

When *n* is large, this *O(n)* procedure can be expensive if performed at runtime. Preprocessing of the vertex data can serve to speed up the process. One simple approach that adds no extra data is based on the fact that only the vertices on the convex hull of the object can contribute to determining the bounding volume shape (Figure 4.6). In the preprocessing step, all *k* vertices on the convex hull of the object would be stored so that they come before all remaining vertices. Then, a tight AABB could be constructed by examining these *k* first vertices only. For general concave volumes this would be a win, but a convex volume, which already has all of its vertices on its convex hull, would see no improvement.

By use of additional, dedicated, precomputed search structures, locating extremal vertices can be performed in *O* (log *n*) time. For instance, the Dobkin-Kirkpatrick hierarchy (described in **Chapter 9**) can be used for this purpose. However, due to the extra memory required by these structures, as well as the overhead in traversing them, they have to be considered overkill in most circumstances. Certainly if tight bounding volumes are that important, tighter bounding volumes than AABBs should be considered.

4.2.5 AABB from Hill-climbing Vertices of the Object Representation

Another way to speed up the AABB realignment process is to use an object representation in which neighboring vertices of a vertex can be found quickly. Such a representation allows the extreme vertices that define the new AABB to be located through simple hill climbing (**Figure 4.7**).

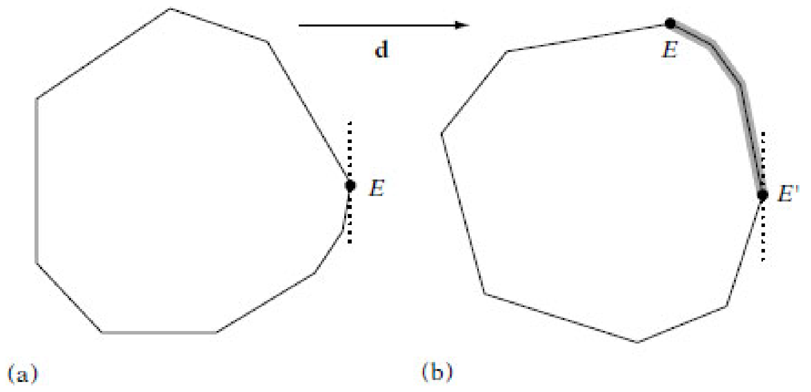

Figure 4.7 (a) The extreme vertex *E* in direction d. (b) After object rotates counterclockwise, the new extreme vertex *E* in direction d can be obtained by hill climbing along the vertex path highlighted in gray.

Instead of keeping track of the minimum and maximum extent values along each axis, six vertex pointers are maintained. Corresponding to the same values as before, these now actually point at the (up to six) extremal vertices of the object along each axis direction. The hill-climbing step now proceeds by comparing the referenced vertices against their neighbor vertices to see if they are still extremal in the same direction as before. Those that are not are replaced with one of their more extreme neighbors and the test is repeated until the extremal vertex in that direction is found. So as not to get stuck in local minima, the hill-climbing process requires objects to be convex. For this reason, hill climbing is performed on precalculated convex hulls of nonconvex objects. Overall, this recalculation of the tight AABB is an expected constant-time operation.

Only having to transform vertices when actually examined by the hill-climbing process greatly reduces computational effort. However, this can be further improved by the realization that only one of the x, y, or *z* components is used in finding the extremal vertex along a given axis. For instance, when finding the extremal point along the +*x* axis only the *x* components of the transformed vertices need to be computed. Hence, the transformational cost is reduced by two-thirds.

Some care must be taken in order to write a robust implementation of this hill-climbing method. Consider an extremal vertex along any axis surrounded by coplanar vertices only. If the object now rotates 180 degrees about any of the remaining two axes, the vertex becomes extremal along the opposite direction along the same axis. However, as it is surrounded by co-planar vertices, the hill-climbing step cannot find a better neighboring vertex and thus terminates with a vertex that is, in fact, the least extremal in the sought direction! A robust implementation must special-case this situation. Alternatively, coplanar vertices can be removed in a preprocessing step, as described in **Chapter 12**. The problem of finding extremal vertices is revisited in **Section 9.5.4**.

4.2.6 AABB Recomputed from Rotated AABB

Last of the four realignment methods, the most common approach is to simply wrap the rotated AABB itself in a new AABB. This produces an approximate rather than a tight AABB. As the resulting AABB is larger than the one that was started with, it is important that the approximate AABB is computed from a rotation of the original local-space AABB. If not, repeated recomputing from the rotated AABB of the previous time step would make the AABB grow indefinitely.

Consider an axis-aligned bounding box *A* affected by a rotation matrix M, resulting in an oriented bounding box *A* at some orientation. The three columns (or rows, depending on what matrix convention is used) of the rotation matrix M give the world-coordinate axes of *A* in its local coordinate frame. (If vectors are column vectors and multiplied on the right of the matrix, then the columns of M are the axes. If instead the vectors are multiplied on the left of the matrix as row vectors, then the rows of M are the axes.)

Say *A* is given using min-max representation and M is a column matrix. The axis-aligned bounding box *B* that bounds *A′* is specified by the extent intervals formed by the projection of the eight rotated vertices of *A*' onto the world-coordinate axes. For, say, the *x* extents of *B*, only the *x* components of the column vectors of M contribute. Therefore, finding the extents corresponds to finding the vertices that produce the minimal and maximal products with the rows of M. Each vertex of *B* is a combination of three transformed min or max values from *A*. The minimum extent value is the sum of the smaller terms, and the maximum extent is the sum of the larger terms. Translation does not affect the size calculation of the new bounding box and can just be added in. For instance, the maximum extent along the *x* axis can be computed as:

```
B.max[0] = max(m[0][0] * A.min[0], m[0][0] * A.max[0])
         + max(m[0][1] * A.min[1], m[0][1] * A.max[1])
         + max(m[0][2] * A.min[2], m[0][2] * A.max[2]) + t[0];
```

Computingan encompassingboundingboxfora rotated AABBusingthe min-max representation can therefore be implemented as follows:

```
// Transform AABB a by the matrix m and translation t,
// find maximum extents, and store result into AABB b.
void UpdateAABB(AABB a, float m[3][3], float t[3], AABB &b)
{
   // For all three axes
   for(inti=0;i<3; i++){
      // Start by adding in translation
      b.min[i] = b.max[i] = t[i];
      // Form extent by summing smaller and larger terms respectively
      for(intj=0;j<3; j++) {
         float e = m[i][j] * a.min[j];
         float f = m[i][j] * a.max[j];
         if(e<f){
            b.min[i] += e;
            b.max[i] += f;
         } else {
            b.min[i] += f;
            b.max[i] += e;
            }
        }
    }
}
```

Correspondingly, the code for the center-radius AABB representation becomes [Arvo90]:

```
  // Transform AABB a by the matrix m and translation t,
  // find maximum extents, and store result into AABB b.
void UpdateAABB(AABB a, float m[3][3], float t[3], AABB &b)
  {
    for(inti=0;i<3; i++){
       b.c[i] = t[i];
       b.r[i] = 0.0f;
       for(intj=0;j<3;
         b.c[i] += m[i][j] * a.c[j];
         b.r[i] += Abs(m[i][j]) * a.r[j];
       }
    }
}
```

Note that computing an AABB from a rotated AABB is equivalent to computing it from a freelyoriented boundingbox. Oriented boundingboxes and their intersection tests will be described in more detail ahead. However, classed between the methods presented here and those to be presented would be the method of storing oriented bounding boxes with the objects but still intersecting them as reconstructed AABBs (as done here). Doing so would require extra memory for storing the orientation matrix. It would also involve an extra matrix-matrix multiplication for combining the rotation matrix of the oriented bounding box with the transformation matrix **M**. The benefit of this solution is that the reconstructed axis-aligned box would be much tighter, starting with an oriented box. The axis-aligned test is also much cheaper than the full-blown test for oriented boxes.

4.3 Spheres

The sphere is another very common bounding volume, rivaling the axis-aligned bounding box in popularity. Like AABBs, spheres have an inexpensive intersection test. Spheres also have the benefit of being rotationally invariant, which means that they are trivial to transform: they simply have to be translated to their new position. Spheres are defined in terms of a center position and a radius:

```
// Region R = { (x, y, z) | (x-c.x)^2 + (y-c.y)^2 + (z-c.z)^2<=r^2 }
struct Sphere {
   Point c; // Sphere center
   float r; // Sphere radius
};
```

At just four components, the bounding sphere is the most memory-efficient bounding volume. Often a preexisting object center or origin can be adjusted to coincide with the sphere center, and only a single component, the radius, need be stored. Computing an optimal bounding sphere is not as easy as computing an optimal axis-aligned bounding box. Several methods of computing bounding spheres are examined in the following sections, in order of increasing accuracy, concluding with an algorithm for computing the minimum bounding sphere. The methods explored for the nonoptimal approximation algorithms remain relevant in that they can be applied to other bounding volumes.

4.3.1 Sphere-sphere Intersection

The overlap test between two spheres is very simple. The Euclidean distance between the sphere centers is computed and compared against the sum of the sphere radii. To avoid an often expensive square root operation, the squared distances are compared. The test looks like this:

```
int TestSphereSphere(Sphere a, Sphere b)
{
   // Calculate squared distance between centers
   Vector d = a.c - b.c;
   float dist2 = Dot(d, d);
   // Spheres intersect if squared distance is less than squared sum of radii
   float radiusSum = a.r + b.r;
   return dist2 <= radiusSum * radiusSum;
}
```

Although the sphere test has a few more arithmetic operations than the AABB test, it also has fewer branches and requires fewer data to be fetched. In modern architectures, the sphere test is probably barely faster than the AABB test. However, the speed of these simple tests should not be a guiding factor in choosing between the two. Tightness to the actual data is a far more important consideration.

4.3.2 Computing a Bounding Sphere

A simple approximative bounding sphere can be obtained by first computing the AABB of all points. The midpoint of the AABB is then selected as the sphere center, and the sphere radius is set to be the distance to the point farthest away from this center point. Note that using the geometric center (the mean) of all points instead of the midpoint of the AABB can give extremely bad bounding spheres for nonuniformly distributed points (up to twice the needed radius). Although this is a fast method, its fit is generally not very good compared to the optimal method.

An alternative approach to computing a simple approximative bounding sphere is described in [Ritter90]. This algorithm tries to find a good initial almost-bounding sphere and then in a few steps improve it until it does bound all points. The algorithm progresses in two passes. In the first pass, six (not necessarily unique) extremal points along the coordinate system axes are found. Out of these six points, the pair of points farthest apart is selected. (Note that these two points do not necessarily correspond to the points defining the longest edge of the AABB of the point set.) The sphere center is now selected as the midpoint between these two points, and the radius is set to be half the distance between them. The code for this first pass is given in the functions **`MostSeparatedPointsOnAABB()`** and **`SphereFromDistantPoints()`** of the following:

```
// Compute indices to the two most separated points of the (up to) six points
// defining the AABB encompassing the point set. Return these as min and max.
void MostSeparatedPointsOnAABB(int &min, int &max, Point pt[], int numPts)
{
  // First find most extreme points along principal axes
  int minx=0,maxx=0,miny=0,maxy=0,minz=0,maxz=0;
  for(inti=1;i< numPts;i++){
    if (pt[i].x < pt[minx].x) minx = i;
    if (pt[i].x > pt[maxx].x) maxx = i;
    if (pt[i].y < pt[miny].y) miny = i;
    if (pt[i].y > pt[maxy].y) maxy = i;
    if (pt[i].z < pt[minz].z) minz = i;
    if (pt[i].z > pt[maxz].z) maxz = i;
  }
    // Compute the squared distances for the three pairs of points
    float dist2x = Dot(pt[maxx] - pt[minx], pt[maxx] - pt[minx]);
    float dist2y = Dot(pt[maxy] - pt[miny], pt[maxy] - pt[miny]);
    float dist2z = Dot(pt[maxz] - pt[minz], pt[maxz] - pt[minz]);
    // Pick the pair (min, max) of points most distant
    min = minx;
    max = maxx;
    if (dist2y > dist2x && dist2y > dist2z) {
       max = maxy;
       min = miny;
    }
    if (dist2z > dist2x && dist2z > dist2y) {
       max = maxz;
        min = minz;
    }
}
void SphereFromDistantPoints(Sphere &s, Point pt[], int numPts)
{
  // Find the most separated point pair defining the encompassing AABB
  int min, max;
  MostSeparatedPointsOnAABB(min, max, pt, numPts);

  // Set up sphere to just encompass these two points
  s.c = (pt[min] + pt[max]) * 0.5f;
  s.r = Dot(pt[max] – s.c, pt[max] – s.c);
  s.r = Sqrt(s.r);
}
```

In the second pass, all points are looped through again. For all points outside the current sphere, the sphere is updated to be the sphere just encompassing the old sphere and the outside point. In other words, the new sphere diameter spans from the outside point to the point on the backside of the old sphere opposite the outside point, with respect to the old sphere center.

```
  // Given Sphere s and Point p, update s (if needed) to just encompass p
  void SphereOfSphereAndPt(Sphere &s, Point &p)
  {
     // Compute squared distance between point and sphere center
     Vector d = p - s.c;
     float dist2 = Dot(d, d);
     // Only update s if point p is outside it
     if (dist2 > s.r * s.r) {
        float dist = Sqrt(dist2);
        float newRadius = (s.r + dist) * 0.5f;
        float k = (newRadius - s.r) / dist;
        s.r = newRadius;
        s.c+=d*k;
    }
}
```

The full code for computing the approximate bounding sphere becomes:

```
void RitterSphere(Sphere &s, Point pt[], int numPts)
{
    // Get sphere encompassing two approximately most distant points
    SphereFromDistantPoints(s, pt, numPts);

    // Grow sphere to include all points
    for (inti=0;i<numPts;i++)
        SphereOfSphereAndPt(s, pt[i]);
}
```

By starting with a better approximation of the true bounding sphere, the resulting sphere could be expected to be even tighter. Using a better starting approximation is explored in the next section.

4.3.3 Bounding Sphere from Direction of Maximum Spread

Instead of finding a pair of distant points using an AABB, as in the previous section, a suggested approach is to analyze the point cloud using statistical methods to find its direction of maximum spread [Wu92]. Given this direction, the two points farthest away from each other when projected onto this axis would be used to determine the center and radius of the starting sphere. Figure 4.8 indicates the difference in spread for two different axes for the same point cloud.

Just as the mean of a set of data values (that is, the sum of all values divided by the number of values) is a measure of the central tendency of the values, *variance* is a measure of their dispersion, or spread. The mean u and the variance σ^2 are given by

$$u = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - u)^2 = \frac{1}{n}\left(\sum_{i=1}^{n}x_i^2\right) - u^2.$$

The square root of the variance is known as the *standard deviation.* For values spread along a single axis, the variance is easily computed as the average of the squared deviation of the values from the mean:

```
// Compute variance of a set of 1D values
float Variance(float x[], int n)
{
   float u = 0.0f;
   for(inti=0;i<n;
      u += x[i];
   u/=n;
   float s2 = 0.0f;
   for(inti=0;i<n; i++)
      s2 += (x[i] - u) * (x[i] - u);
    return s2 / n;
}
```

Usually there is no obvious direct interpretation of variance and standard deviation. They are, however, important as comparative measures. For two variables, the *covariance* measures their tendency to vary together. It is computed as the average of products of deviation of the variable values from their means. For multiple variables, the covariance of the data is conventionally computed and expressed as a matrix, the *covariance matrix* (also referred to as the variance-covariance or dispersion matrix).

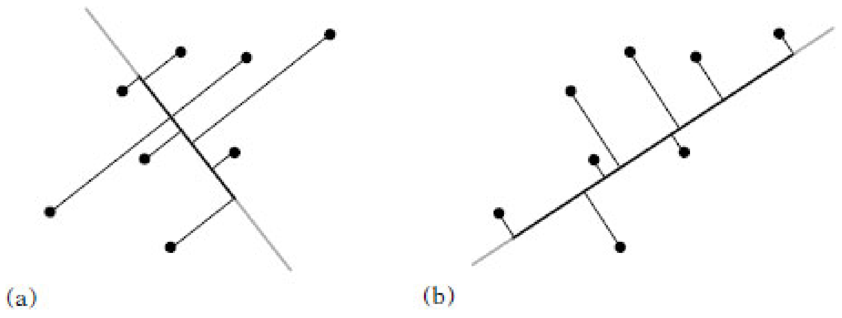

Figure 4.8 The same point cloud projected onto two different axes. In (a) the spread on the axis is small. In (b) the spread is much larger. A bounding sphere can be determined from the axis for which the projected point set has the maximum spread.

The covariance matrix C = [c_{ij}] for a collection of *n* points $P_1, P_2, \ldots, P_n$ is given by

$$c_{ij} = \frac{1}{n} \sum_{k=1}^{n} \left(P_{k,i} - u_i\right)\left(P_{k,j} - u_j\right),$$

or equivalently by

$$c_{ij} = \frac{1}{n} \left(\sum_{k=1}^{n} P_{k,i} P_{k,j} \right) - u_i u_j.$$

The u_i (and u_j) term is the mean of the i-th coordinate value of the points, given by

$$u_i = \frac{1}{n} \sum_{k=1}^{n} P_{k,i}.$$

Informally, to see how covariance works, consider the first covariance formula. When two variables tend to deviate in the same direction from their respective means, the product,

$$\left(P_{k,i} - u_i\right)\left(P_{k,j} - u_j\right),$$

will be positive more often than negative. If the variables tend to deviate in different directions, the product will be negative more often than positive. The sum of these products identifies how the variables co-vary. When implemented using single-precision floats, the former of the two covariance formulas tends to produce results that are more accurate by retaining more bits of precision. Using double precision, there is typically little or no difference in the results. The following code implements the first formula:

```
void CovarianceMatrix(Matrix33 &cov, Point pt[], int numPts)
{
   float oon = 1.0f / (float)numPts;
   Point c = Point(0.0f, 0.0f, 0.0f);
   float e00, e11, e22, e01, e02, e12;

   // Compute the center of mass (centroid) of the points
   for(inti=0;i<numPts;i++)
      c += pt[i];
   c *= oon;

   // Compute covariance elements
   e00 = ell = e22 = e01 = e02 = e12 = 0.0f;
```

```
    for(inti=0;i< numPts;i++){
       // Translate points so center of mass is at origin
       Point p = pt[i] - c;
       // Compute covariance of translated points
       e00 += p.x * p.x;
       e11 += p.y * p.y;
       e22 += p.z * p.z;
       e01 += p.x * p.y;
       e02 += p.x * p.z;
       e12 += p.y * p.z;
    }
    // Fill in the covariance matrix elements
    cov[0][0] = e00 * oon;
    cov[1][1] = e11 * oon;
    cov[2][2] = e22 * oon;
    cov[0][1] = cov[1][0] = e01 * oon;
    cov[0][2] = cov[2][0] = e02 * oon;
    cov[1][2] = cov[2][1] = e12 * oon;
 }
```

Once the covariance matrix has been computed, it can be decomposed in a manner that reveals more about the principal directions of the variance. This decomposition is performed by computing the eigenvalues and eigenvectors of the matrix. The relationship between these is such that the eigenvector associated with the largest magnitude eigenvalue corresponds to the axis along which the point data has the largest variance. Similarly, the eigenvector associated with the smallest magnitude eigenvalue is the axis along which the data has the least variance. Robustly finding the eigenvalues and eigenvectors of a matrix is a nontrivial task in general. Typically, they are found using some (iterative) numerical technique (for which a good source is [Golub96]).

By definition, the covariance matrix is always symmetric. As a result, it decomposes into real (rather than complex) eigenvalues and an orthonormal basis of eigenvectors. For symmetric matrices, a simpler decomposition approach can be used. For a moderate-size matrix, as here, the Jacobi method works quite well. The intricate details of the Jacobi method are beyond the scope of this book. Briefly, however, the algorithm performs a number of transformation steps to the given input matrix. Each step consists of applying a rotation to the matrix, bringing the matrix closer and closer to a diagonal matrix (all elements zero, except along the diagonal). When the matrix is diagonal, the elements on the diagonal are the eigenvalues. While this is done, all rotations are also concatenated into another matrix. Upon exit, this matrix will contain the eigenvectors. Ideally, this decomposition should be performed in double-precision arithmetic to minimize numerical errors. The following code for the Jacobi method is based on the presentation in [Golub96]. First is a subroutine for assisting in computing the rotation matrix.

```
// 2-by-2 Symmetric Schur decomposition. Given an n-by-n symmetric matrix
// and indices p, q such that 1 <= p < q <= n, computes a sine-cosine pair
// (s, c) that will serve to form a Jacobi rotation matrix.
//
// See Golub, Van Loan, Matrix Computations, 3rd ed, p428
void SymSchur2(Matrix33 &a, int p, int q, float &c, float &s)
{
   if (Abs(a[p][q]) > 0.0001f) {
      float r = (a[q][q] - a[p][p]) / (2.0f * a[p][q]);
      float t;
      if (r >= 0.0f)
         t = 1.0f / (r + Sqrt(1.0f + r*r));
      else
        t = -1.0f / (-r + Sqrt(1.0f + r*r));
      c = 1.0f / Sqrt(1.0f + t*t);
      s=t*c;
   } else {
      c = 1.0f;
      s = 0.0f;
   }
}
```

Given this support function, the full Jacobi method is now implemented as:

```
// Computes the eigenvectors and eigenvalues of the symmetric matrix A using
// the classic Jacobi method of iteratively updating A as A = J^T*A*J,
// where J = J(p, q, theta) is the Jacobi rotation matrix.
//
// On exit, v will contain the eigenvectors, and the diagonal elements
// of a are the corresponding eigenvalues.
//
// See Golub, Van Loan, Matrix Computations, 3rd ed, p428
void Jacobi(Matrix33 &a, Matrix33 &v)
{
   int i, j, n, p, q;
   float prevoff, c, s;
```

```
    Matrix33 J, b, t;

    // Initialize v to identify matrix
    for(i=0;i<3;
       v[i][0] = v[i][1] = v[i][2] = 0.0f;
       v[i][i] = 1.0f;
    }

    // Repeat for some maximum number of iterations
    const int MAXITERATIONS = 50;
    for (n = 0; n < MAXITERATIONS; n++) {
       // Find largest off-diagonal absolute element a[p][q]
       p=0;q=1;
       for(1=0;i<3;
           for(j=0;j<3; j++){
              if (i == j) continue;
              if (Abs(a[i][j]) > Abs(a[p][q])) {
                  p=i;
                  q = j;
              }
           }
        }

       // Compute the Jacobi rotation matrix J(p, q, theta)
       // (This code can be optimized for the three different cases of rotation)
       SymSchur2(a, p, q, c, s);
       for(i=0;i<3; i++){
          J[i][0] = J[i][1] = J[i][2] = 0.0f;
          J[i][i] = 1.0f;
       }
       J[p][p] = c; J[p][q] = s;
       J[q][p] = -s; J[q][q] = c;

       // Cumulate rotations into what will contain the eigenvectors
       v = v * J;

       // Make ‘a’ more diagonal, until just eigenvalues remain on diagonal
       a = (J.Transpose() * a) * J;

       // Compute “norm” of off-diagonal elements
       float off = 0.0f;
       for(1=0;i<3;
           for(j = 0;j<3;
               if (i == j) continue;
               off += a[i][j] * a[i][j];
           }
        }
        /* off = sqrt(off); not needed for norm comparison */

        // Stop when norm no longer decreasing
        if(n>2&&off>= prevoff)
           return;

           prevoff = off;
        }
     }
```

For the particular 3 x 3 matrix used here, instead of applying a general approach such as the Jacobi method the eigenvalues could be directly computed from a simple cubic equation. The eigenvectors could then easily be found through, for example, Gaussian elimination. Such an approach is described in [Cromwell94]. Given the previously defined functions, computing a sphere from the two most distant points (according to spread) now looks like:

```
void EigenSphere(Sphere &eigSphere, Point pt[], int numPts)
{

   Matrix33 m, v;

   // Compute the covariance matrix m
   CovarianceMatrix(m, pt, numPts);
   // Decompose it into eigenvectors (in v) and eigenvalues (in m)
   Jacobi(m, v);

   // Find the component with largest magnitude eigenvalue (largest spread)
   Vector e;    int maxc = 0;
   float maxf, maxe = Abs(m[0][0]);
   if ((maxf = Abs(m[1][1])) > maxe) maxc = 1, maxe = maxf;
   if ((maxf = Abs(m[2][2])) > maxe) maxc = 2, maxe = maxf;
   e[0] = v[0][maxc];
   e[1] = v[1][maxc];
   e[2] = v[2][maxc];

   // Find the most extreme points along direction ‘e’
   int imin, imax;
   ExtremePointsAlongDirection(e, pt, numPts, &imin, &imax);
   Point minpt = pt[imin];
   Point maxpt = pt[imax];

   float dist = Sqrt(Dot(maxpt - minpt, maxpt - minpt));
```

```
    eigSphere.r = dist * 0.5f;
    eigSphere.c = (minpt + maxpt) * 0.5f;
  }
```

The modified full code for computing the approximate bounding sphere becomes:

```
void RitterEigenSphere(Sphere &s, Point pt[], int numPts)
{
   // Start with sphere from maximum spread
   EigenSphere(s, pt, numPts);

   // Grow sphere to include all points
   for(inti=0;i<numPts;i++)
     SphereOfSphereAndPt(s, pt[i]);
}
```

The type of covariance analysis performed here is commonly used for dimension reduction and statistical analysis of data, and is known as *principal component analysis* (PCA). Further information on PCA can be found in [**Jolliffe02**]. The eigenvectors of the covariance matrix can also be used to orient an oriented bounding box, as described in **Section 4.4.3**.

4.3.4 Bounding Sphere Through Iterative Refinement

The primary idea behind the algorithm described in **Section 4.3.2** is to start with a quite good, slightly underestimated, approximation to the actual smallest sphere and then grow it until it encompasses all points. Given a better initial sphere, the final sphere can be expected to be better as well. Consequently, it is hardly surprising that the output of the algorithm can very effectively be used to feed itself in an iterative manner. The resulting sphere of one iteration is simply shrunk by a small amount to make it an underestimate for the next iterative call.

```
void RitterIterative(Sphere &s, Point pt[], int numPts)
{
   const int NUMITER = 8;
   RitterSphere(s, pt, numPts);
   Sphere s2 = s;
   for(intk=0;k< NUM_ITER;k++){
      // Shrink sphere somewhat to make it an underestimate (not bound)
      s2.r = s2.r * 0.95f;
```

```
        // Make sphere bound data again
        for(inti=0;i<numPts;i++){
            // Swap pt[i] with pt[j], where j randomly from interval [i+1, numPts-1]
            DoRandomSwap();
            SphereOfSphereAndPt(s2, pt[i]);
        }

        // Update s whenever a tighter sphere is found
        if (s2.r < s.r) s = s2;
    }
}
```

To further improve the results, the points are considered at random, rather than in the same order from iteration to iteration. The resulting sphere is usually much better than that produced by Wu's method (described in the previous section), at the cost of a few extra iterations over the input data. If the same iterative approach is applied to Wu's algorithm, the results are comparable. As with all iterative hill-climbing algorithms of this type (such as gradient descent methods, simulated annealing, or TABU search), the search can get stuck in local minima, and an optimal result is not guaranteed. The returned result is often very nearly optimal, however. The result is also very robust.

4.3.5 The Minimum Bounding Sphere

A sphere is uniquely defined by four (non co-planar) points. Thus, a brute-force algorithm for computing the minimum bounding sphere for a set of points is to consider all possible combinations of four (then three, then two) points, computing the smallest sphere through these points and keeping the sphere if it contains all other points. The kept sphere with the smallest radius is then the minimum bounding sphere. This brute-force algorithm has a complexity of $O(n^5)$ and is therefore not practical. Fortunately, the problem of computing the minimum bounding sphere for a set of points has been well studied in the field of computational geometry, and a randomized algorithm that runs in expected linear time has been given by [Welzl91].

Assume a minimum bounding sphere S has been computed for a point set P.Ifa new point Q is added to P, then only if Q lies outside S does S need to be recomputed. It is not difficult to see that Q must lie on the boundary of the new minimum bounding sphere for the point set $P \cup \{Q\}$. Welzl's algorithm is based on this observation, resulting in a recursive algorithm. It proceeds by maintaining both the set of input points and a *set of support*, which contains the points from the input set that must lie on the boundary of the minimum sphere. The following code fragment outlines Welzl's algorithm:

```
Sphere WelzlSphere(Point pt[], unsigned int numPts, Point sos[], unsigned int numS
{
  // if no input points, the recursion has bottomed out. Now compute an
  // exact sphere based on points in set of support (zero through four points)
  if (numPts == 0) {
    switch (numSos) {
    case 0: return Sphere();
    case 1: return Sphere(sos[0]);
    case 2: return Sphere(sos[0], sos[1]);
    case 3: return Sphere(sos[0], sos[1], sos[2]);
    case 4: return Sphere(sos[0], sos[1], sos[2], sos[3]);
    }
  }
  // Pick a point at "random" (here just the last point of the input set)
  int index = numPts - 1;
  // Recursively compute the smallest bounding sphere of the remaining points
  Sphere smallestSphere = WelzlSphere(pt, numPts - 1, sos, numSos); // (*)
  // If the selected point lies inside this sphere, it is indeed the smallest
  if(PointInsideSphere(pt[index], smallestSphere))
    return smallestSphere;
  // Otherwise, update set of support to additionally contain the new point
  sos[numSos] = pt[index];
  // Recursively compute the smallest sphere of remaining points with new s.o.s.
  return WelzlSphere(pt, numPts - 1, sos, numSos + 1);
}
```

Although the two recursive calls inside the function make the function appear expensive, Welzl showed that assuming points are removed from the input set at random the algorithm runs in expected linear time. Note that, as presented, the first recursive call (marked with an asterisk in the code) is likely to cause stack overflow for inputs of more than a few thousand points. Slight changes to the code avoid this problem, as outlined in [Gärtner99]. A full implementation is given in [Capens01]. Also available on the Internet is a more intricate implementation, part of the Computational Geometry Algorithms Library (CGAL). Writing a robust implementation of Welzl's algorithm requires that the four support functions for computing exact spheres from one through four points must correctly deal with degenerate input, such as collinear points.

Welzl's algorithm can be applied to computing both bounding circles and higher dimensional balls. It does not, however, directly extend to computing the minimum sphere bounding a set of spheres. An algorithm for the latter problem is given in [Fischer03]. Having covered spheres in detail, we now turn our attention to bounding boxes of arbitrary orientation.

4.4 Oriented Bounding Boxes (OBBs)

An oriented bounding box (OBB) is a rectangular block, much like an AABB but with an arbitrary orientation. There are many possible representations for an OBB: as a collection of eight vertices, a collection of six planes, a collection of three slabs (a pair of parallel planes), a corner vertex plus three mutually orthogonal edge vectors, or a center point plus an orientation matrix and three halfedge lengths. The latter is commonly the preferred representation for OBBs, as it allows for a much cheaper OBB-OBB intersection test than do the other representations. This test is based on the separating axis theorem, which is discussed in more detail in **Chapter 5**.

```
// Region R = { x|x = c+r*u[0]+s*u[1]+t*u[2]}, |r|<=e[0], |s|<=e[1], |t|<=e[2]
struct OBB {
    Point c; // OBB center point
    Vector u[3]; // Local x-, y-, and z-axes
    Vector e; // Positive halfwidth extents of OBB along each axis
};
```

At 15 floats, or 60 bytes for IEEE single-precision floats, the OBB is quite an expensive bounding volume in terms of memory usage. The memory requirements could be lowered by storing the orientation not as a matrix but as Euler angles or as a quaternion, using three to four floating-point components instead of nine. Unfortunately, for an OBB-OBB intersection test these representations must be converted back to a matrix for use in the effective separating axis test, which is a very expensive operation. A good compromise therefore may be to store just two of the rotation matrix axes and compute the third from a cross product of the other two at test time. This relatively cheap CPU operation saves three floating-point components, resulting in a 20% memory saving.

4.4.1 OBB-OBB Intersection

Unlike the previous bounding volume intersection tests, the test for overlap between two oriented bounding boxes is surprisingly complicated. At first, it seems a test to see if either box that is fully outside a face of the other would suffice. In its simplest form, this test could be performed by checking if the vertices of box *A* are all on the outside of the planes defined by the faces of box *B*, and vice versa. However, although this test works in 2D it does not work correctly in 3D. It fails to deal with, for example, the case in which *A* and *B* are almost meeting edge to edge, the edges perpendicular to each other. Here, neither box is fully outside any one face of the other. Consequently, the simple test reports them as intersecting even though they are not. Even so, the simple test may still be useful. Although it is not always correct, it is conservative in that it never fails to detect a collision. Only in some cases does it incorrectly report separated boxes as overlapping. As such, it can serve as a pretest for a more expensive exact test.

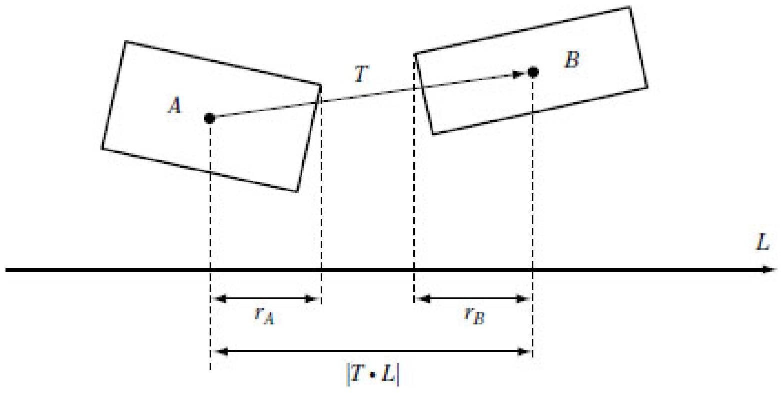

Figure 4.9 Two OBBs are separated if for some axis *L* the sum of their projected radii is less than the distance between their projected centers.

An exact test for OBB-OBB intersection can be implemented in terms of what is known as the separating axis test. This test is discussed in detail in Chapter 5, but here it is sufficient to note that two OBBs are separated if, with respect to some axis L, the sum of their projected radii is less than the distance between the projection of their center points (as illustrated in Figure 4.9). That is, if

$$|T \cdot L| > r_A + r_B.$$

For OBBs it is possible to show that at most 15 of these separating axes must be tested to correctly determine the OBB overlap status. These axes correspond to the three coordinate axes of *A*, the three coordinate axes of

B, and the nine axes perpendicular to an axis from each. If the boxes fail to overlap on any of the 15 axes, they are not intersecting. If no axis provides this early out, it follows that the boxes must be overlapping.

The number of operations in the test can be reduced by expressing *B* in the coordinate frame of A. If **t** is the translation vector from *A* to *B* and **R** = [r_{ij}] (the rotation matrix bringing *B* into *A*'s coordinate frame), the tests that must be performed for the different axes *L* are summarized in Table 4.1.

Table 4.1 The 15 separating axis tests needed to determine OBB-OBB intersection. Superscripts indicate which OBB the value comes from.

L	$\lvert T \cdot L \rvert$	r_A	r_B
$\mathbf{u}_0^A$	$\lvert \mathbf{t}_0 \rvert$	$\mathbf{e}_0^A$	$\mathbf{e}_0^B \lvert r_{00} \rvert + \mathbf{e}_1^B \lvert r_{01} \rvert + \mathbf{e}_2^B \lvert r_{02} \rvert$
$\mathbf{u}_1^A$	$\lvert \mathbf{t}_1 \rvert$	$\mathbf{e}_1^A$	$\mathbf{e}_0^B \lvert r_{10} \rvert + \mathbf{e}_1^B \lvert r_{11} \rvert + \mathbf{e}_2^B \lvert r_{12} \rvert$
$\mathbf{u}_2^A$	$\lvert \mathbf{t}_2 \rvert$	$\mathbf{e}_2^A$	$\mathbf{e}_0^B \lvert r_{20} \rvert + \mathbf{e}_1^B \lvert r_{21} \rvert + \mathbf{e}_2^B \lvert r_{22} \rvert$
$\mathbf{u}_0^B$	$\lvert \mathbf{t}_0 r_{00} + \mathbf{t}_1 r_{10} + \mathbf{t}_2 r_{20} \rvert$	$\mathbf{e}_0^A \lvert r_{00} \rvert + \mathbf{e}_1^A \lvert r_{10} \rvert + \mathbf{e}_2^A \lvert r_{20} \rvert$	$\mathbf{e}_0^B$
$\mathbf{u}_1^B$	$\lvert \mathbf{t}_0 r_{01} + \mathbf{t}_1 r_{11} + \mathbf{t}_2 r_{21} \rvert$	$\mathbf{e}_0^A \lvert r_{01} \rvert + \mathbf{e}_1^A \lvert r_{11} \rvert + \mathbf{e}_2^A \lvert r_{21} \rvert$	$\mathbf{e}_1^B$
$\mathbf{u}_2^B$	$\lvert \mathbf{t}_0 r_{02} + \mathbf{t}_1 r_{12} + \mathbf{t}_2 r_{22} \rvert$	$\mathbf{e}_0^A \lvert r_{02} \rvert + \mathbf{e}_1^A \lvert r_{12} \rvert + \mathbf{e}_2^A \lvert r_{22} \rvert$	$\mathbf{e}_2^B$
$\mathbf{u}_0^A \times \mathbf{u}_0^B$	$\lvert \mathbf{t}_2 r_{10} - \mathbf{t}_1 r_{20} \rvert$	$\mathbf{e}_1^A \lvert r_{20} \rvert + \mathbf{e}_2^A \lvert r_{10} \rvert$	$\mathbf{e}_1^B \lvert r_{02} \rvert + \mathbf{e}_2^B \lvert r_{01} \rvert$
$\mathbf{u}_0^A \times \mathbf{u}_1^B$	$\lvert \mathbf{t}_2 r_{11} - \mathbf{t}_1 r_{21} \rvert$	$\mathbf{e}_1^A \lvert r_{21} \rvert + \mathbf{e}_2^A \lvert r_{11} \rvert$	$\mathbf{e}_0^B \lvert r_{02} \rvert + \mathbf{e}_2^B \lvert r_{00} \rvert$
$\mathbf{u}_0^A \times \mathbf{u}_2^B$	$\lvert \mathbf{t}_2 r_{12} - \mathbf{t}_1 r_{22} \rvert$	$\mathbf{e}_1^A \lvert r_{22} \rvert + \mathbf{e}_2^A \lvert r_{12} \rvert$	$\mathbf{e}_0^B \lvert r_{01} \rvert + \mathbf{e}_1^B \lvert r_{00} \rvert$
$\mathbf{u}_1^A \times \mathbf{u}_0^B$	$\lvert \mathbf{t}_0 r_{20} - \mathbf{t}_2 r_{00} \rvert$	$\mathbf{e}_0^A \lvert r_{20} \rvert + \mathbf{e}_2^A \lvert r_{00} \rvert$	$\mathbf{e}_1^B \lvert r_{12} \rvert + \mathbf{e}_2^B \lvert r_{11} \rvert$
$\mathbf{u}_1^A \times \mathbf{u}_1^B$	$\lvert \mathbf{t}_0 r_{21} - \mathbf{t}_2 r_{01} \rvert$	$\mathbf{e}_0^A \lvert r_{21} \rvert + \mathbf{e}_2^A \lvert r_{01} \rvert$	$\mathbf{e}_0^B \lvert r_{12} \rvert + \mathbf{e}_2^B \lvert r_{10} \rvert$
$\mathbf{u}_1^A \times \mathbf{u}_2^B$	$\lvert \mathbf{t}_0 r_{22} - \mathbf{t}_2 r_{02} \rvert$	$\mathbf{e}_0^A \lvert r_{22} \rvert + \mathbf{e}_2^A \lvert r_{02} \rvert$	$\mathbf{e}_0^B \lvert r_{11} \rvert + \mathbf{e}_1^B \lvert r_{10} \rvert$
$\mathbf{u}_2^A \times \mathbf{u}_0^B$	$\lvert \mathbf{t}_1 r_{00} - \mathbf{t}_0 r_{10} \rvert$	$\mathbf{e}_0^A \lvert r_{10} \rvert + \mathbf{e}_1^A \lvert r_{00} \rvert$	$\mathbf{e}_1^B \lvert r_{22} \rvert + \mathbf{e}_2^B \lvert r_{21} \rvert$
$\mathbf{u}_2^A \times \mathbf{u}_1^B$	$\lvert \mathbf{t}_1 r_{01} - \mathbf{t}_0 r_{11} \rvert$	$\mathbf{e}_0^A \lvert r_{11} \rvert + \mathbf{e}_1^A \lvert r_{01} \rvert$	$\mathbf{e}_0^B \lvert r_{22} \rvert + \mathbf{e}_2^B \lvert r_{20} \rvert$
$\mathbf{u}_2^A \times \mathbf{u}_2^B$	$\lvert \mathbf{t}_1 r_{02} - \mathbf{t}_0 r_{12} \rvert$	$\mathbf{e}_0^A \lvert r_{12} \rvert + \mathbf{e}_1^A \lvert r_{02} \rvert$	$\mathbf{e}_0^B \lvert r_{21} \rvert + \mathbf{e}_1^B \lvert r_{20} \rvert$

This test can be implemented as follows:

```
int TestOBBOBB(OBB &a, OBB &b)
{
    float ra, rb;
    Matrix33 R, AbsR;

    // Compute rotation matrix expressing b in a’s coordinate frame
    for (int i =0;i<3; i++)
       for (int j = 0;j<3; j++)
          R[i][j] = Dot(a.u[i], b.u[j]);
```

```
// Compute translation vector t
Vector t = b.c - a.c;
// Bring translation into a’s coordinate frame
t = Vector(Dot(t, a.u[0]), Dot(t, a.u[1]), Dot(t, a.u[2]));

// Compute common subexpressions. Add in an epsilon term to
// counteract arithmetic errors when two edges are parallel and
// their cross product is (near) null (see text for details)
for (int i=0;i<3; i++)
   for (int j=0;j<3; j++)
       AbsR[i][j] = Abs(R[i][j]) + EPSILON;

// Test axes L = A0, L = A1, L = A2
for (int i=0;i<3; i++) {       ra = a.e[i];
    rb = b.e[0] * AbsR[i][0] + b.e[1] * AbsR[i][1] + b.e[2] * AbsR[i][2];
    if (Abs(t[i]) > ra + rb) return 0;
}

// Test axes L = B0, L = B1, L = B2
for (int i=0;i<3; i++) {
    ra = a.e[0] * AbsR[0][i] + a.e[1] * AbsR[1][i] + a.e[2] * AbsR[2][i];
    rb = b.e[i];
    if (Abs(t[0] * R[0][i] + t[1] * R[1][i] + t[2] * R[2][i]) > ra + rb) retur
}

// Test axis L = A0 x B0
ra = a.e[1] * AbsR[2][0] + a.e[2] * AbsR[1][0];
rb = b.e[1] * AbsR[0][2] + b.e[2] * AbsR[0][1];
if (Abs(t[2] * R[1][0] - t[1] * R[2][0]) > ra + rb) return 0;

// Test axis L = A0 x B1
ra = a.e[1] * AbsR[2][1] + a.e[2] * AbsR[1][1];
rb = b.e[0] * AbsR[0][2] + b.e[2] * AbsR[0][0];
if (Abs(t[2] * R[1][1] - t[1] * R[2][1]) > ra + rb) return 0;

// Test axis L = A0 x B2
ra = a.e[1] * AbsR[2][2] + a.e[2] * AbsR[1][2];
rb = b.e[0] * AbsR[0][1] + b.e[1] * AbsR[0][0];
if (Abs(t[2] * R[1][2] - t[1] * R[2][2]) > ra + rb) return 0;

// Test axis L = A1 x B0
ra = a.e[0] * AbsR[2][0] + a.e[2] * AbsR[0][0];
rb = b.e[1] * AbsR[1][2] + b.e[2] * AbsR[1][1];
if (Abs(t[0] * R[2][0] - t[2] * R[0][0]) > ra + rb) return 0;
```

```
    // Test axis L = A1 x B1
    ra = a.e[0] * AbsR[2][1] + a.e[2] * AbsR[0][1];
    rb = b.e[0] * AbsR[1][2] + b.e[2] * AbsR[1][0];
    if (Abs(t[0] * R[2][1] - t[2] * R[0][1]) > ra + rb) return 0;

    // Test axis L = A1 x B2
    ra = a.e[0] * AbsR[2][2] + a.e[2] * AbsR[0][2];
    rb = b.e[0] * AbsR[1][1] + b.e[1] * AbsR[1][0];
    if (Abs(t[0] * R[2][2] - t[2] * R[0][2]) > ra + rb) return 0;

    // Test axis L = A2 x B0
    ra = a.e[0] * AbsR[1][0] + a.e[1] * AbsR[0][0];
    rb = b.e[1] * AbsR[2][2] + b.e[2] * AbsR[2][1];
    if (Abs(t[1] * R[0][0] - t[0] * R[1][0]) > ra + rb) return 0;

    // Test axis L = A2 x B1
    ra = a.e[0] * AbsR[1][1] + a.e[1] * AbsR[0][1];
    rb = b.e[0] * AbsR[2][2] + b.e[2] * AbsR[2][0];
    if (Abs(t[1] * R[0][1] - t[0] * R[1][1]) > ra + rb) return 0;

    // Test axis L = A2 x B2
    ra = a.e[0] * AbsR[1][2] + a.e[1] * AbsR[0][2];
    rb = b.e[0] * AbsR[2][1] + b.e[1] * AbsR[2][0];
    if (Abs(t[1] * R[0][2] - t[0] * R[1][2]) > ra + rb) return 0;

    // Since no separating axis is found, the OBBs must be intersecting
    return 1;
}
```

To make the OBB-OBB test as efficient as possible, it is important that the axes are tested in the order given in **Table 4.1**. The first reason for using this order is that by testing three orthogonal axes first there is little spatial redundancy in the tests, and the entire space is quickly covered. Second, with the setup given here, where *A* is transformed to the origin and aligned with the coordinate system axes, testing the axes of *A* is about half the cost of testing the axes of B. Although it is not done here, the calculations of **`R`** and **`AbsR`** should be interleaved with the first three tests, so that they are not unnecessarily performed in their entirety when the OBB test exits in one of the first few *if* statements.

If OBBs are used in applications in which they often tend to have one axis aligned with the current world up, for instance, when traveling on ground, it is worthwhile special-casing these “vertically aligned” OBBs. This simplification allows for a much faster intersection test that only involves testing four separating axes in addition to a cheap test in the vertical direction.

In some cases, performing just the first 6 of the 15 axis tests may result in faster results overall. In empirical tests, [Bergen97] found that the last 9 tests in the OBB overlap code determine nonintersection about 15% of the time. As perhaps half of all queries are positive to start with, omitting these 9 tests results in false positives about 6 to 7% of the time. When the OBB test is performed as a pretest for an exact test on the bounded geometry, this still leaves the test conservative and no collisions are missed.

4.4.2 Making the Separating-axis Test Robust

A very important issue overlooked in several popular treatments of the separating-axis theorem is the robustness of the test. Unfortunately, any code implementing this test must be very carefully crafted to work as intended. When a separating axis is formed by taking the cross product of an edge from each bounding box there is a possibility these edges are parallel. As a result, their cross product is the null vector, all projections onto this null vector are zero, and the sum of products on each side of the axis inequality vanishes. Remaining is the comparison $0 > 0$. In the perfect world of exact arithmetic mathematics, this expression would trivially evaluate to false. In reality, any computer implementation must deal with inaccuracies introduced by the use of floating-point arithmetic.

For the optimized inequalities presented earlier, the case of parallel edges corresponds to only the zero elements of the rotation matrix $\mathbf{R}$ being referenced. Theoretically, this still results in the comparison $0 > 0$. In practice, however, due to accumulation of errors the rotation matrix will not be perfectly orthonormal and its zero elements will not be exactly zero. Thus, the sum of products on both sides of the inequality will also not be zero, but some small error quantity. As this accumulation of errors can cause either side of the inequality to change sign or shift in magnitude, the result will be quite random. Consequently, if the inequality tests are not very carefully performed these arithmetic errors could lead to the (near) null vector incorrectly being interpreted as a separating axis. Two overlapping OBBs therefore could be incorrectly reported as nonintersecting.

As the right-hand side of the inequalities should be larger when two OBBs are interpenetrating, a simple solution to the problem is to add a small epsilon value to the absolute values of the matrix elements occurring on the right-hand side of the inequalities. For near-zero terms, this epsilon term will be dominating and axis tests corresponding to (near) parallel edges are thus made disproportionately conservative. For other, nonzero cases, the small epsilon term will simply disappear. Note that as the absolute values of the components of a rotation matrix are bounded to the range [0, 1] using a fixed-magnitude epsilon works fine regardless of the sizes of the boxes involved. The robustness of the separating-axis test is revisited in **Chapter 5**.

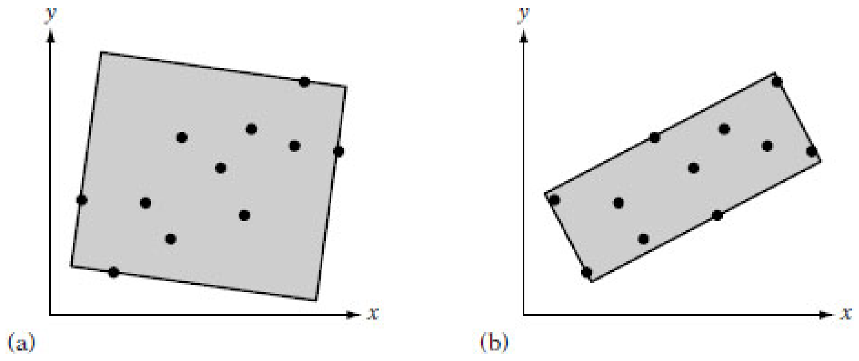

Figure 4.10 (a) A poorly aligned and (b) a well-aligned OBB.

4.4.3 Computing a Tight OBB

Computing tight-fitting oriented bounding boxes is a difficult problem, made worse by the fact that the volume difference between a poorly aligned and a well-aligned OBB can be quite large (**Figure 4.10**). There exists an algorithm for calculating the minimum volume bounding box of a polyhedron, presented in [**O'Rourke85**]. The key observation behind the algorithm is that given a polyhedron either one face and one edge or three edges of the polyhedron will be on different faces of its bounding box. Thus, these edge and face configurations can be searched in a systematic fashion, resulting in an $O(n^3)$ algorithm. Although it is an interesting theoretical result, unfortunately the algorithm is both too complicated and too slow to be of much practical value.

Two other theoretical algorithms for computing near approximations of the minimum-volume bounding box are presented in [**Barequet99**]. However, the authors admit that these algorithms are probably too difficult to implement and would be impractical even so, due to the large constant-factor overhead in the algorithms. Thus, with the currently available theoretical algorithms of little practical use OBBs must be computed using either approximation or brute-force methods.

A simpler algorithm offered in [Barequet99] provides a coarse approximation of the optimal OBB for a point set by first computing the minimum AABB of the set. From the point set, a pair of points on the two parallel box sides farthest apart are selected to determine the length direction of the OBB. The set of points is then projected onto the plane perpendicular to the OBB length direction. The same procedure is now applied again, only this time computing the minimum axis-aligned rectangle, with points on the two parallel sides farthest apart determining a second axis for the OBB. The third OBB axis is the perpendicular to the first two axes. Although this algorithm is very easy to code, in practice bounding boxes much closer to optimal can be obtained through other algorithms of similar complexity, as described in the following.

For long and thin objects, an OBB axis should be aligned with the direction of the objects. For a flat object, an OBB axis should be aligned with the normal of the flat object. These directions correspond to the principal directions of the objects, and the principal component analysis used in Section 4.3.3 can be used here.

Computing bounding boxes based on covariance of model vertices generally works satisfactorily for models whose vertices are uniformly distributed over the model space. Unfortunately, the influence of internal points often bias the covariance and can make the OBB take on any orientation regardless of the extremal points. For this reason, all methods for computing bounding volumes based on weighting vertex positions should ideally be avoided. It is sufficient to note that the defining features (center, dimensions, and orientation) of a minimum bounding volume are all independent of clustering of object vertices. This can easily be seen by considering adding (or taking away) extra vertices off-center, inside or on the boundary, of a bounding volume. These actions do not affect the defining features of the volume and therefore should not affect its calculation. However, adding extra points in this manner changes the covariance matrix of the points, and consequently any OBB features directly computed from the matrix. The situation can be improved by considering just extremal points, using only those points on the convex hull of the model. This eliminates the internal points, which can no longer misalign the OBB. However, even though all remaining points are extremal the resulting OBB can still be arbitrarily bad due to point distribution. A clustering of points will still bias an axis toward the cluster. In other words, using vertices alone simply cannot produce reliable covariance matrices.

A suggested solution is to use a continuous formulation of covariance, computing the covariance across the entire face of the primitives [Gottschalk00]. The convex hull should still be used for the calculation. If not, small outlying geometry would extend the bounding box, but not contribute enough significance to align the box properly. In addition, interior geometry would still bias the covariance. If the convex hull is already available, this algorithm is $O(n)$. If the convex hull must be computed, it is $O(n \log n)$.

Given n triangles $(p_k, q_k, r_k), 0 \le k < n$, in the convex hull, the covariance matrix is given by

$$C_{ij} = \left(\frac{1}{a_H} \sum_{0 \le k < n} \frac{a_k}{12} (9m_{k,i}m_{k,j} + p_{k,i}p_{k,j} + q_{k,i}q_{k,j} + r_{k,i}r_{k,j}) \right) - m_{H,i}m_{H,j},$$

where $a_k = ||(q_k - p_k) \times (r_k - p_k)||/2$ is the area and $m_k = (p_k + q_k + r_k)/3$ is the centroid of triangle k.

The total area of the convex hull is given by

$$a_H = \sum_{0 \le k < n} a_k,$$

and the centroid of the convex hull,

$$m_H = \frac{1}{a_H} \sum_{0 \le k < n} a_k m_k,$$

is computed as the mean of the triangle centroids weighted by their area. The i and j subscripts indicate which coordinate component is taken (that is, x, y, or z). Code for this calculation can be found in the publicly available collision detection package RAPID. A slightly different formulation of this covariance matrix is given in [Eberly01], and code is available on the companion CD-ROM for this book.

The method just described treats the polyhedron as a hollow body, computing the covariance matrix from the surface areas. A related method treating the polyhedron as a solid body is described in [Mirtich96a].

Given an assumed uniform density polyhedron, Mirtich's polyhedral mass property algorithm integrates over the volume of the polyhedron, computing its 3 × 3 inertia tensor (also known as the inertia or mass matrix). The eigenvectors of this symmetric matrix are called the principal axes of inertia, and the eigenvalues the principal moments of inertia. Just as with the covariance matrices before, the Jacobi method can be used to extract these axes, which in turn can then serve as the orientation matrix of an OBB. Detailed pseudocode for computing the inertia matrix is given in Mirtich's article. A public domain implementation in C is also available for download on the Internet. Mirtich's article is revisited in [Eberly03], in which a more computationally efficient approach is derived and for which pseudocode is provided.

Note that neither covariance-aligned nor inertia-aligned oriented bounding boxes are optimal. Consider an object *A* and its associated OBB *B*. Let *A* be expanded by adding some geometry to it, but staying within the bounds of *B*. For both methods, this would in general result in a different OBB *B*' for the new object *A*'. By construction, *B* and *B*' both cover the two objects *A* and *A*'. However, as the dimensions of the two OBBs are in the general case different, one OBB must be suboptimal.

4.4.4 Optimizing PCA-based OBBs

As covariance-aligned OBBs are not optimal, it is reasonable to suspect they could be improved through slight modification. For instance, perhaps the OBB could be rotated about one of its axes to find the orientation for which its volume is smallest. One improved approach to OBB fitting is to align the box along just one principal component. The remaining two directions are determined from the computed minimum-area bounding rectangle of the projection of all vertices onto the perpendicular plane to the selected axis. Effectively, this method determines the best rotation about the given axis for producing the smallest-volume OBB. This approach was suggested in [Barequet96], in which three different methods were investigated.

- All-principal-components box
- Max-principal-component box
- Min-principal-component box

The all-principal-components box uses all three principal components to align the OBB, and is equivalent to the method presented in the previous section. For the max-principal-component box, the eigenvector corresponding to the largest eigenvalue is selected as the length of the OBB. Then all points are projected onto a plane perpendicular to that direction. The projection is followed by computing the minimum-area bounding rectangle of the projected points, determining the remaining two directions of the OBB. Finally, the min-principal-component box selects the shortest principal component as the initial direction of the OBB, and then proceeds as in the previous method. Based on empirical results, [**Barequet96**] conclude that the min-principal-component method performs best. A compelling reason max-principal-component does not do better is also given: as the maximum principal component is the direction with the maximum variance, it will contribute the longest possible edge and is therefore likely to produce a larger volume.

A local minimum volume can be reached by iterating this method on a given starting OBB. The procedure would project all vertices onto the plane perpendicular to one of the directions of the OBB, updating the OBB to align with the minimum-area bounding rectangle of the projection. The iterations would be repeated until no projection (along any direction of the OBB) gives an improvement, at which point the local minimum has been found. This method serves as an excellent optimization of boxes computed through other methods, such as Mirtich's, when performed as a preprocessing step.

Remaining is the problem of computing the minimum-area bounding rectangle of a set of points in the plane. The key insight here is a result from the field of computational geometry. It states that a minimum-area bounding rectangle of a convex polygon has (at least) one side collinear with an edge of the polygon [**Freeman75**].

Therefore, the minimum rectangle can trivially be computed by a simple algorithm. First, the convex hull of the point set is computed, resulting in a convex polygon. Then, an edge at a time of the polygon is considered as one direction of the bounding box. The perpendicular to this direction is obtained, and all polygon vertices are projected onto these two axes and the rectangle area is computed. When all polygon edges have been tested, the edge (and its perpendicular) giving the smallest area determines the directions of the minimum-area bounding rectangle. For each edge considered, the rectangle area is computed in $O(n)$ time, for a total complexity of $O(n^2)$ for the algorithm as a whole. This algorithm can be implemented as follows:

```
// Compute the center point, ‘c’, and axis orientation, u[0] and u[1], of
// the minimum area rectangle in the xy plane containing the points pt[].
float MinAreaRect(Point2D pt[], int numPts, Point2D &c, Vector2D u[2])
{
    float minArea = FLT_MAX;

    // Loop through all edges; j trails i by 1, modulo numPts
    for (int i = 0, j= numPts-1;i < numPts; j=i, i++){
        // Get current edge e0 (e0x, e0y), normalized
        Vector2D e0 = pt[i] - pt[j];
        e0 /= Length(e0);

        // Get an axis e1 orthogonal to edge e0
        Vector2D e1 = Vector2D(-e0.y, e0.x); // = Perp2D(e0)

        // Loop through all points to get maximum extents
        float min0 = 0.0f, min1 = 0.0f, max0 = 0.0f, max1 = 0.0f;
        for (int k=0; k < numPts; k++){
            // Project points onto axes e0 and e1 and keep track
            // of minimum and maximum values along both axes
            Vector2D d = pt[k] - pt[j];
            float dot = Dot2D(d, e0);
            if (dot < min0) min0 = dot;
            if (dot > max0) max0 = dot;
            dot = Dot2D(d, e1);
            if (dot < min1) min1 = dot;
            if (dot > max1) max1 = dot;
        }
        float area = (max0 - min0) * (max1 - min1);

       // If best so far, remember area, center, and axes
       if (area < minArea) {
           minArea = area;
           c = pt[j] + 0.5f * ((min0 + max0) * e0 + (min1 + max1) * e1);
           u[0] = e0; u[1] = e1;
       }
    }
    return minArea;
 }
```

The minimum-area bounding rectangle of a convex polygon can also be computed in $O(n \log n)$ time, using the method of *rotating calipers* [Toussaint83]. The rotating calipers algorithm starts out bounding the polygon by four lines through extreme points of the polygon such that the lines determine a rectangle. At least one line is chosen to be coincident with an edge of the polygon. For each iteration of the algorithm, the lines are simultaneously rotated clockwise about their supporting points until a line coincides with an edge of the polygon. The lines now form a new bounding rectangle around the polygon. The process is repeated until the lines have been rotated by an angle of 90 degrees from their original orientation. The minimum-area bounding rectangle corresponds to the smallest-area rectangle determined by the lines over all iterations. The time complexity of the algorithm is bounded by the cost of computing the convex hull. If the convex hull is already available, the rotating calipers algorithm is $O(n)$.

4.4.5 Brute-force OBB Fitting

The last approach to OBB fitting considered here is simply to compute the OBB in a brute-force manner. One way to perform brute-force fitting is to parameterize the orientation of the OBB in some manner. The space of orientations is sampled at regular intervals over the parameterization and the best OBB over all sampled rotations is kept. The OBB orientation is then refined by sampling the interval in which the best OBB was found at a higher subinterval resolution. This hill-climbing approach is repeated with smaller and smaller interval resolutions until there is little or no change in orientation for the best OBB.

For each tested coordinate system, computing the candidate OBB requires the transformation of all vertices into the coordinate system. Because this transformation is expensive, the search should exit as soon as the candidate OBB becomes worse than the currently best OBB. In that it is cheap to compute and has a relatively good fit, a PCA-fitted OBB provides a good initial guess, increasing the chances of an early out during point transformation [Miettinen02a]. To further increase the chance of an early out, the (up to) six extreme points determining the previous OBB should be the first vertices transformed. In [Miettinen02b] it is reported that over 90% of the tests are early-exited using this optimization. Brute-force fitting of OBBs generally results in much tighter OBBs than those obtained through PCA-based fitting.

The hill-climbing approach just described considers many sample points in the space of orientation before updating the currently best OBB. The optimization-based OBB-fitting method described in [Lahanas00] hill climbs the search space one sample at a time, but employs a multisample technique to aid the optimizer escape from local minima.

4.5 Sphere-swept Volumes

After spheres and boxes, it is natural to consider cylinders as bounding volumes. Unfortunately, when the mathematics is worked out it turns out that the overlap test for cylinders is quite expensive, making them less attractive as bounding volumes. However, if a cylinder is fitted with spherical end caps the resulting capped cylinder volume becomes a more attractive bounding volume. Let the cylinder be described by the points A and B (forming its medial axis) and a radius r. The capped cylinder would be the resulting volume from sweeping a sphere of radius r along the line segment AB. This volume is part of a family of volumes, extensions of the basic sphere.

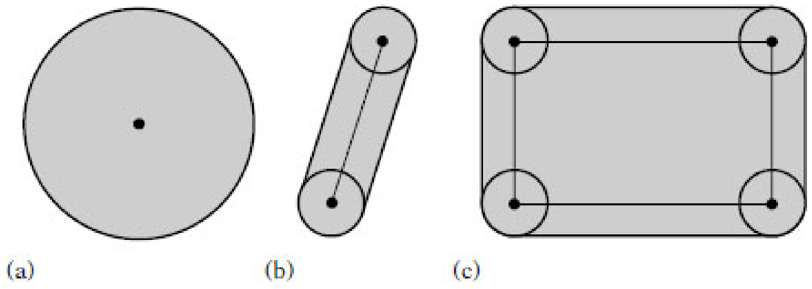

Figure 4.11 (a) A sphere-swept point (SSP). (b) A sphere-swept line (SSL). (c) A sphere-swept rectangle (SSR).

Recall that the test between two spheres computes the distance between the two center points, comparing the result against the sum of the radii (squaring the quantities to avoid an expensive square root). By replacing the sphere center points with arbitrary inner primitives or medial structures, new bounding volumes can be obtained. The resultant volumes are equivalent to sweeping the inner primitive with a sphere of radius r (or, technically, forming the Minkowski sum of the sphere and the primitive). As such, this entire family of bounding volumes is therefore collectively referred to as *sphere-swept volumes* (SSVs). All points within a distance r from the inner medial structure belong to the SSV. Three types of sphere-swept volumes are illustrated in Figure 4.11.

Following the sphere overlap test, the intersection test for two SSVs simply becomes calculating the (squared) distance between the two inner primitives and comparing it against the (squared) sum of their combined radii. The cost of sphere-swept tests is completely determined by the cost of the distance function. To make the distance test as inexpensive as possible, the inner primitives are usually limited to points, line segments, or rectangles. The resulting SSVs — sphere-swept points (SSPs), sphere-swept lines (SSLs), and sphere-swept rectangles (SSRs) — are commonly referred to, respectively, as *spheres*, *capsules*, and *lozenges*. The latter looks like an OBB with rounded corners and edges. Capsules are also referred to as *capped cylinders* or *spherocylinders*. The data structures for capsules and lozenges can be defined as follows:

```
// Region R = { x | (x - [a + (b - a)*t])^2<=r } ,0<=t<=1
struct Capsule {
    Point a;    // Medial line segment start point
    Point b;    // Medial line segment endpoint
    float r;    // Radius
};

// Region R = { x | (x - [a + u[0]*s + u[1]*t])^2<=r }, 0 <= s, t <= 1
struct Lozenge {
    Point a;    // Origin
    Vector u[2]; // The two edges axes of the rectangle
    float r;    // Radius
};
```

Due to similarity in shape, lozenges can be a viable substitute for OBBs. In situations of close proximity, the lozenge distance computation becomes less expensive than the OBB separating-axis test.

4.5.1 Sphere-swept Volume Intersection

By construction, all sphere-swept volume tests can be formulated in the same way. First, the distance between the inner structures is computed. Then this distance is compared against the sum of the radii. The only difference between any two types of sphere-swept tests is in the calculation used to compute the distance between the inner structures of the two volumes. A useful property of sphere-swept volumes is that the distance computation between the inner structures does not rely on the inner structures being of the same type. Mixed-type or hybrid tests can therefore easily be constructed. Two tests are presented in the following: the sphere-capsule and capsule-capsule tests.

```
int TestSphereCapsule(Sphere s, Capsule capsule)
{
    // Compute (squared) distance between sphere center and capsule line segment
    float dist2 = SqDistPointSegment(capsule.a, capsule.b, s.c);
    // If (squared) distance smaller than (squared) sum of radii, they collide
    float radius = s.r + capsule.r;
    return dist2 <= radius * radius;
}

int TestCapsuleCapsule(Capsule capsule1, Capsule capsule2)
{
  // Compute (squared) distance between the inner structures of the capsules
  float s, t;
  Point c1, c2;
  float dist2 = ClosestPtSegmentSegment(capsule1.a,
                      capsule1.b, capsule2.a, capsule2.b, s, t, c1, c2);
  // If (squared) distance smaller than (squared) sum of radii, they collide
  float radius = capsule1.r + capsule2.r;
  return dist2 <= radius * radius;
}
```

The functions **`SqDistPointSegment()`** and **`ClosestPtSegmentSegment()`** used here are found in **Chapter 5** (**Sections 5.1.2.1** and **5.1.9**, respectively). A distance test for SSRs (and thus, by reduction, also for SSLs and SSPs) based on halfspace tests is given in [**Larsen99**] and [**Larsen00**].

4.5.2 Computing Sphere-swept Bounding Volumes

The machinery needed for computing SSVs has been described in previous sections. For instance, by computing the principal axes a capsule can be fitted by using the longest axis as the capsule length. The next longest axis determines the radius. Alternatively, the length can also be fit using a least-square approach for fitting a line to a set of points. For SSRs, all three axes would be used, with the shortest axis forming the rectangle face normal. See [**Eberly01**], [**Larsen99**], and [**Larsen00**] for further detail.

4.6 Halfspace Intersection Volumes

With the notable exception of spheres, most bounding volumes are convex polyhedra. All of these polyhedral bounding volumes are representable as the intersection of a set of halfspaces, wherein the halfspace dividing planes coincides with the sides of the bounding volume. For instance, AABBs and OBBs are both the intersection of six halfspaces. A tetrahedron is the intersection of four halfspaces — the smallest number of halfspaces needed to form a closed volume. Generally, the more halfspaces used the better the resulting intersection volume can fit an object. If the bounded object is polyhedral, the tightest possible convex bounding volume is the convex hull of the object. In this case, the number of faces on the hull serves as a practical upper limit on how many halfspaces are needed to form the bounding volume. Convex hulls are important bounding volumes, and an in-depth treatment of collision detection algorithms for convex hulls is given in **Chapter 9**.

Although convex hulls form the tightest bounding volumes, they are not necessarily the best choice of bounding volume. Some drawbacks of convex hulls include their being expensive and difficult to compute, taking large amounts of memory to represent, and potentially being costly to operate upon. By limiting the number of halfspaces used in the intersection volume, several simpler alternative bounding volumes can be formed. A few variants are described in the sections that follow.

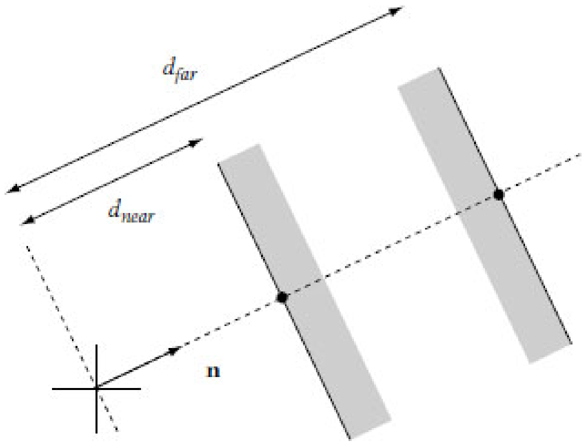

Figure 4.12 A slab is the infinite region of space between two planes, defined by a normal n and two signed distances from the origin.

4.6.1 Kay-Kajiya Slab-based Volumes

First introduced by Kay and Kajiya for speeding up ray-object intersection tests, Kay-Kajiya volumes are a family of many-sided parallelepipedal bounding volumes based on the intersection of slabs [Kay86]. A slab is the infinite region of space between two parallel planes (Figure 4.12). This plane set is represented by a unit vector **n** (the plane-set normal) and two scalar values giving the signed distance from the origin (along **n**) for both planes.

```
// Region R = { (x, y, z) | dNear <= a*x + b*y + c*z <= dFar }
struct Slab {
    float n[3]; // Normal n = (a, b, c)
    float dNear; // Signed distance from origin for near plane (dNear)
    float dFar; // Signed distance from origin for far plane (dFar)
};
```

To form a bounding volume, a number of normals are chosen. Then, for each normal, pairs of planes are positioned so that they bound the object on both its sides along the direction of the normal. For polygonal objects, the positions of the planes can be found by computing the dot product of the normal and each object vertex. The minimum and maximum values are then the required scalar values defining the plane positions.

To form a closed 3D volume, at least three slabs are required. Both AABBs and OBBs are examples of volumes formed as the intersection of three slabs. By increasing the number of slabs, slab-based bounding volumes can be made to fit the convex hulls of objects arbitrarily tightly.

For the original application of ray intersection, a quick test can be based on the fact that a parameterized ray intersects a slab if and only if it is simultaneously inside all (three) slabs for some interval, or equivalently if the intersection of the intervals for ray-slab overlap is nonzero. With pairs of planes sharing the same normal, calculations can be shared between the two ray-plane intersection tests, improving test performance.

Although slab-based volumes allow for fast ray intersection tests, they cannot easily be used as-is in object-object intersection testing. However, by sharing normals across all objects it is possible to perform fast object-object tests (explored in the next section).

4.6.2 Discrete-orientation Polytopes (*k*-DOPs)

Based on the same idea as the Kay–Kajiya slab-based volumes are the volumes known as *discrete-orientation polytopes* (*k*-DOPs) or *fixed-direction hulls* (FDHs), suggested by [Konečný97] and [Klosowski98]. (Although the latter is perhaps a more descriptive term, the former is more commonly used and is adopted here.) These *k*-DOPs are convex polytopes, almost identical to the slab-based volumes except that the normals are defined as a fixed set of axes shared among all *k*-DOP bounding volumes. The normal components are typically limited to the set {−1,0,1}, and the normals are not normalized. These normals make computation of *k*-DOPs cheaper, which is important because *k*-DOPs must be dynamically realigned. By sharing the normals among all objects, *k*-DOP storage is very cheap. Only the min-max intervals for each axis must be stored. For instance, an 8-DOP becomes:

```
struct DOP8 {
  float min[4]; // Minimum distance (from origin) along axes 0 to 3
  float max[4]; // Maximum distance (from origin) along axes 0 to 3
};
```

A 6-DOP commonly refers to polytopes with faces aligned along the six directions (±1,0,0), (0, ±1,0), and (0,0, ±1). This 6-DOP is of course an AABB, but the AABB is just a special-case 6-DOP; any oriented bounding box could also be described as a 6-DOP. An 8-DOP has faces aligned with the eight directions (±1, ±1, ±1) and a 12-DOP with the 12 directions (±1, ±1,0), (±1, 0, ±1), and (0, ±1, ±1). An example of a 2D 8-DOP is shown in Figure 4.13.

A 14-DOP is defined using the combined directions of the 6-DOP and 8-DOP. The 18-DOP, 20-DOP, and 26-DOP are formed in a similar way. The 14-DOP corresponds to an axis-aligned box with the eight corners cut off. The 18-DOP is an AABB wherein the 12 edges have been cut off. The 18-DOP is also referred to as a *tribox* [Crosnier99].

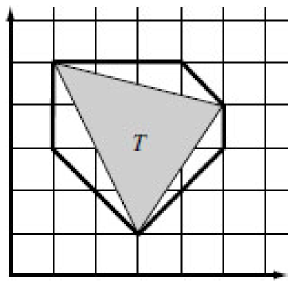

Figure 4.13 8-DOP for triangle (3, 1), (5, 4), (1, 5) is {1, 1, 4, −4, 5, 5, 9, 2} for axes (1, 0), (0, 1), (1, 1), (1, -1).

Note that the *k*-DOP is not just any intersection of slabs, but the tightest slabs that form the body. For example, the triangle (0, 0), (1, 0), (0, 1) can be expressed as several slab intersections, but there is only one *k*-DOP describing it. If and only if the slab planes have a common point with the volume formed by the intersection of the slabs are the slabs defining a *k*-DOP. This tightness criterion is important, as the overlap test for *k*-DOPs does not work on the intersection volume but solely on the slab definitions. Without the given restriction, the overlap test would be quite ineffective.

Compared to oriented bounding boxes, the overlap test for *k*-DOPs is much faster (about an order of magnitude), even for large-numbered DOPs, thanks to the fixed-direction planes. In terms of storage, the same amount of memory required for an OBB roughly corresponds to a 14-DOP. *k*-DOPs have a (probable) tighter fit than OBBs, certainly as *k* grows and the *k*-DOP resembles the convex hull. For geometry that does not align well with the axes of the *k*-DOP, OBBs can still be tighter. OBBs will also perform better in situations of close proximity.

The largest drawback to *k*-DOPs is that even if the volumes are rarely colliding the *k*-DOPs must still be updated, or "tumbled." As this operation is expensive, whenever possible the tumbling operation should be avoided. This is usually done by pretesting the objects using bounding spheres (not performing the *k*-DOP test if the sphere test fails). In general, *k*-DOPs perform best when there are few dynamic objects being tested against many static objects (few *k*-DOPs have to update) or when the same object is involved in several tests (the cost of updating the *k*-DOP is amortized over the tests).

A slightly different *k*-DOP is examined in [Zachmann00]. Here, the *k*-DOP axes are selected using a simulated annealing process in which *k* points are randomly distributed on a unit sphere, with the provision that for each point P, $-P$ is also in the set. A repelling force among points is used as the annealing temperature.

4.6.3 *k*-DOP-*k*-DOP Overlap Test

Due to the fixed set of axes being shared among all objects, intersection detection for *k*-DOPs is similar to and no more difficult than testing two AABBs for overlap. The test simply checks the $k/2$ intervals for overlap. If any pair of intervals do not overlap, the *k*-DOPs do not intersect. Only if all pairs of intervals are overlapping are the *k*-DOPs intersecting.

```
int TestKDOPKDOP (KDOP &a, KDOP &b, int k)
{
  // Check if any intervals are non-overlapping, return if so
  for (int i=0;i<k/2; i++)
      if (a.min[i] > b.max[i] || a.max[i] < b.min[i])
         return 0;

  // All intervals are overlapping, so k-DOPs must intersect
  return 1;
}
```

As with oriented bounding boxes, the order in which axes are tested is likely to have an effect on performance. As intervals along “near” directions are likely to have similar overlap status, consecutive interval tests preferably should be performed on largely different (perpendicular) directions. One way to achieve this is to preprocess the order of the global axes using a simple greedy algorithm. Starting with any axis to test first, it would order all axes so that the successor to the current axis is the one whose dot product with the current axis is as close to zero as possible.

4.6.4 Computing and Realigning *k*-DOPs

Computing the *k*-DOP for an object can be seen as a generalization of the method for computing an AABB, much as the overlap test for two *k*-DOPs is really a generalization of the AABB-AABB overlap test. As such, a *k*-DOP is simply computed from the projection spread of the object vertices along the defining axes of the *k*-DOP. Compared to the AABB calculation, the only differences are that the vertices have to be projected onto the axes and there are more axes to consider. The restriction to keep axis components in the set $\{-1,0,1\}$ makes a hardcoded function for computing the *k*-DOP less expensive than a general function for arbitrary directions, as the projection of vertices onto these axes now involves at most three additions. For example, an 8-DOP is computed as follows:

```
// Compute 8-DOP for object vertices v[] in world space
// using the axes (1,1,1), (1,1, -1), (1, -1,1) and (-1,1,1)
void ComputeDOP8(Point v[], int numPts, DOP8 &dop8)
{
    // Initialize 8-DOP to an empty volume
    dop8.min[0] = dop8.min[1] = dop8.min[2] = dop8.min[3] = FLT_MAX;
    dop8.max[0] = dop8.max[1] = dop8.max[2] = dop8.max[3] = -FLT_MAX;

    // For each point, update 8-DOP bounds if necessary
    float value;
    for (int i=0;i< numPts; i++){
       // Axis 0 = (1,1,1)
       value = v[i].x + v[i].y + v[i].z;
       if (value < dop8.min[0]) dop8.min[0] = value;
       else if (value > dop8.max[0]) dop8.max[0] = value;

       // Axis 1 = (1,1,-1)
       value = v[i].x + v[i].y - v[i].z;
       if (value < dop8.min[1]) dop8.min[1] = value;
       else if (value > dop8.max[1]) dop8.max[1] = value;

       // Axis 2 = (1,-1,1)
       value = v[i].x - v[i].y + v[i].z;
       if (value < dop8.min[2]) dop8.min[2] = value;
       else if (value > dop8.max[2]) dop8.max[2] = value;

       // Axis 3 = (-1,1,1)
       value = -v[i].x + v[i].y + v[i].z;
       if (value < dop8.min[3]) dop8.min[3] = value;
       else if (value > dop8.max[3]) dop8.max[3] = value;
    }
}
```

Although *k*-DOPs are invariant under translation, a rotation leaves the volume unaligned with the predefined axes. As with AABBs, *k*-DOPs must therefore be realigned whenever the volume rotates. A simple solution is to recompute the *k*-DOP from scratch, as described. However, because re-computing the *k*-DOP involves transforming the object vertices into the new space this becomes expensive when the number of object vertices is large. A more effective realignment approach is to use a hill-climbing method similar to that described in **Section 4.2.5** for computing AABBs. The only difference is that instead of keeping track of six references to vertices of the convex hull the hill climber would for a *k*-DOP now keep track of *k* vertex references, one for each facial direction of the *k*-DOP. If frame-to-frame coherency is high and objects rotate by small amounts between frames, tracking vertices is a good approach. However, the worst-case complexity of this method is $O(n^2)$ and it can perform poorly when coherency is low. Hill climbing results in a tight bounding volume.

Another, approximative, approach is based on computing and storing the vertices of each *k*-DOP for its initial local orientation at preprocess time. Then at runtime the *k*-DOP is recomputed from these vertices and transformed into world space by the current orientation matrix. This is equivalent to the AABB method described in **Section 4.2.6**.

The vertex set of the *k*-DOP in its initial orientation can be computed with the help of a duality transformation mapping. Here the dual mapping of a plane $ax + by + cz = 1$ is the point (a, b, c) and vice versa. Let the defining planes of the *k*-DOP be expressed as plane equations. Then by computing the convex hull of the dual of these planes the faces (edges and vertices) of the convex hull maps into the vertices (edges and faces) of the intersection of the original planes when transformed back under the duality mapping. For this duality procedure to work, the *k*-DOP must be translated to contain the origin, if it is not already contained in the *k*-DOP. For volumes formed by the intersection of halfspaces, a point in the volume interior can be obtained through linear programming (see **Section 9.4.1**). Another, simpler, option is to compute an interior point using the *method of alternating projection* (MAP). This method starts with an arbitrary point in space. Looping over all halfspaces, in arbitrary order, the point is updated by projecting it to the bounding hyperplane of the current halfspace whenever it lies outside the halfspace. Guaranteed to converge, the loop over all halfspaces is repeated until the point lies inside all halfspaces. If the starting point lies outside the intersection volume, as is likely, the resulting point will lie on the boundary of the intersection volume, and specifically on one of the bounding hyperplanes. (If the point lies inside the volume, the point itself is returned by the method.) A point interior to the volume is obtained by repeating the method of alternating projections with different starting points until a second point, on a different bounding hyperplane, is obtained. The interior point is then simply the midpoint of the two boundary points. MAP may converge very slowly for certain inputs. However, for the volumes typically used as bounding volumes, slow convergence is usually not a problem. For more information on MAP, see [Deutsch01]. For more information on duality (or polarity) transformations see, for example, [Preparata85], [O'Rourke98], or [Berg00].

A simpler way of computing the initial vertex set is to consider all combinations of three non co-planar planes from the input set of *k*-DOP boundary planes. Each such set of three planes intersects in a point (Section 5.4.5 describes how this point is computed). After all intersection points have been computed, those that lie in front of one or more of the *k*-DOP boundary planes are discarded. The remaining points are then the vertices of the *k*-DOP.

k-DOPs can also be realigned using methods based on linear programming, as described in [Konečný97] and [Konečný98]. A more elaborate realignment strategy is presented in [Fünfzig03]. Linear programming is described in Section 9.4.

4.6.5 Approximate Convex Hull Intersection Tests

It is easy to test separation between two convex polygons (for example, using the rotating calipers method mentioned in Section 4.4.4). Unfortunately, the problem is not as simple for polytopes. Accurate methods for this problem are discussed in Chapter 9. In many situations, however, fully accurate polytope collision detection might be neither necessary nor desired. By relaxing the test to allow approximate solutions, it is possible to obtain both simpler and faster methods.

One such approach maintains both the defining planes and vertices of each convex hull. To test two hulls against each other, the vertices of each hull are tested against the planes of the other to see if they lie fully outside any one plane. If they do, the hulls are not intersecting. If neither set of vertices is outside any face of the other hull, the hulls are conservatively considered overlapping. In terms of the separating-axis test, this corresponds to testing separation on the face normals of both hulls, but not the edge-edge combinations of both.

Another approach is simply to replace the vertex set with a set of spheres. The spheres are chosen so that their union approximates the convex hull. Now the test proceeds by testing spheres (instead of vertices) against the planes. The idea is that compared to vertex tests fewer sphere tests must be performed. Although this test is faster, it is also less accurate. To improve accuracy while keeping the set size down, the set of spheres is often hand optimized.

As with *k*-DOPs, testing can be sped up by ordering the stored planes to make successive planes as perpendicular to each other as possible. Similarly, to avoid degenerate behavior due to clustering the vertices (or spheres) can be randomly ordered. Bounding the vertex set with a sphere and testing the sphere against the plane before testing all vertices often allow for early outs.

Compared to other bounding volume tests, these tests are still relatively expensive and are typically preceded by a cheaper bounding volume test (such as a sphere-sphere test) to avoid hull-testing objects that are sufficiently far apart to not be intersecting. The coherency methods presented in Chapter 9 are also useful for minimizing the number of hull tests that have to be performed.

4.7 Other Bounding Volumes

In addition to the bounding volumes covered here, many other types of volumes have been suggested as bounding volumes. These include cones [Held97], [Eberly02], cylinders [Held97], [Eberly00], [Schömer00], spherical shells [Krishnan98], ellipsoids [Rimon92], [Wang01], [Choi02], [Wang02], [Chien03], and zonotopes [Guibas03]. Cones, cylinders, and ellipsoids are self-explanatory. Spherical shells are the intersection of the volume between two concentric spheres and a cone with its apex at the sphere center. Zonotopes are centrally symmetric polytopes of certain properties. These shapes have not found widespread use as bounding volumes, in part due to having expensive intersection tests. For this reason, they are not covered here further.

It should be noted that whereas ellipsoid-ellipsoid is an expensive intersection test, tests of ellipsoids against triangles and other polygons can be transformed into testing a sphere against a skewed triangle by applying a nonuniform scaling to the coordinate space. Thus, ellipsoids are feasible bounding volumes for certain sets of tests.

4.8 Summary

Bounding volumes are simple geometric shapes used to encapsulate one or more objects of greater geometrical complexity. Most frequently, spheres and boxes are used as bounding volumes. If a really tight fit is required, slab volumes or convex hulls may be used. Bounding volumes are used as early overlap rejection tests, before more expensive tests are performed on the geometry enclosed within them. As discussed in **Section 4.1**, there are trade-offs involved in the selection of bounding volume shapes. By using bounding volumes of tighter fit, the chance of early rejection increases, but at the same time the bounding volume test becomes more expensive and the storage requirement for the bounding volume increases. Typically, bounding volumes are computed in a preprocessing step and, as necessary, transformed with the bounded objects at runtime to match the objects' movements.

In addition to detailing the most common bounding volumes and how to compute them, this chapter described how to perform homogeneous intersection tests (between volumes of the same type). These tests were meant as a teaser toward **Chapter 5**, which in considerable detail covers (heterogeneous) intersection tests and distance calculations between primitive geometrical shapes, such as lines and line segments, spheres, boxes, triangles, polygons, and polyhedra.