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.

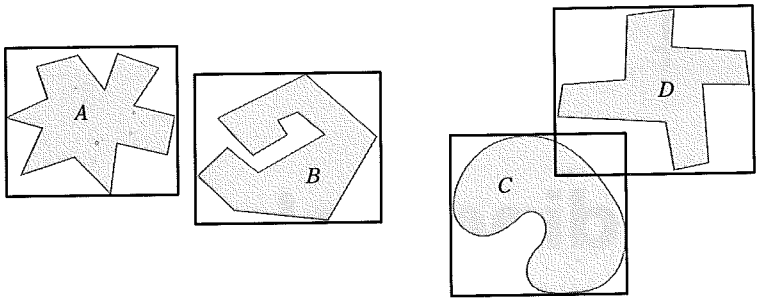

Figure 4.1 The bounding volumes of *A* and *B* do not overlap, and thus *A* and *B* cannot be intersecting. Intersection between *C* and *D* cannot be ruled out because their bounding volumes overlap.

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

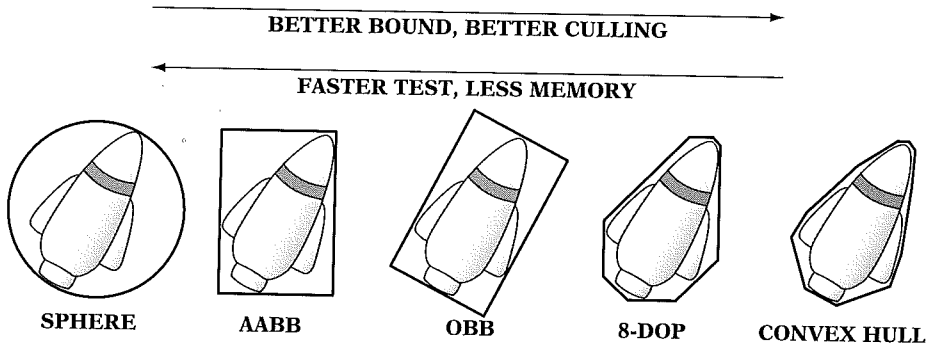

Figure 4.2 Types of bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), eight-direction discrete orientation polytope (8-DOP), and convex hull.

point inclusion, ray intersection with the volume, and intersection with planes and polygons.

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.

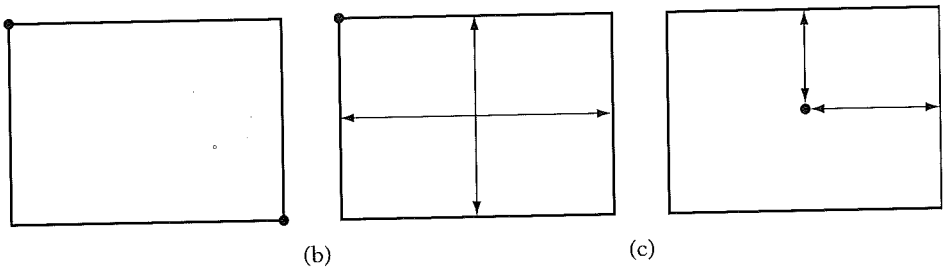

Figure 4.3 The three common AABB representations: (a) min-max, (b) min-widths, and (c) center-radius.

There are three common representations for AABBs (Figure 4.3). One is 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+dz}
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.

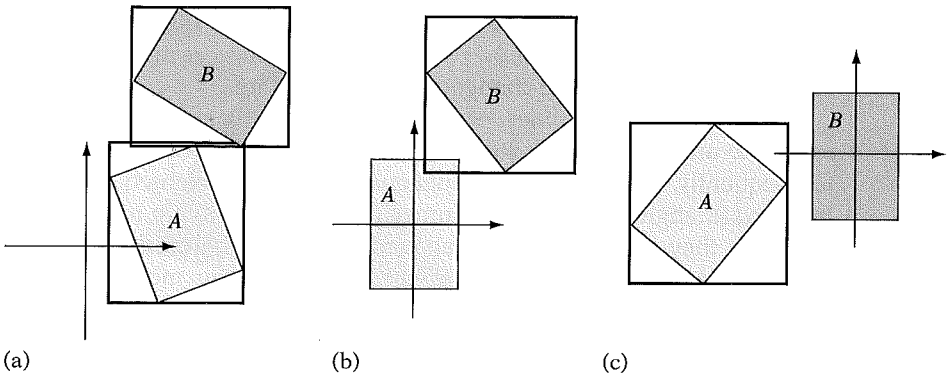

Figure 4.4 (a) AABBs *A* and *B* in world space. (b) The AABBs in the local space of *A*. (c) The AABBs in the local space of *B*.

4.2.2 Computing and Updating AABBs

Bounding volumes are usually specified in the local model space of the objects they bound (which may be 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 bounding box, 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

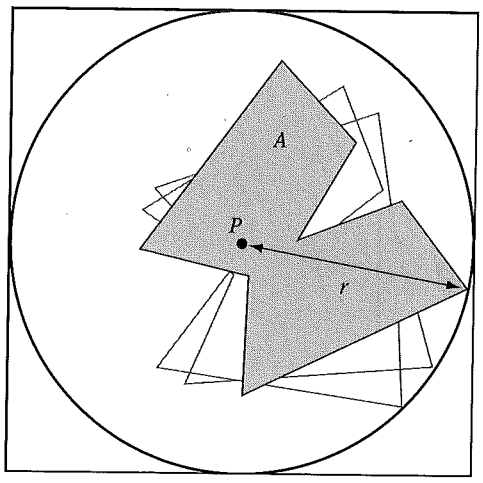

Figure 4.5 AABB of the bounding sphere that fully contains object *A* under an arbitrary orientation.

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:

```
// 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 *imax)
{
    float minproj = FLT_MAX, maxproj = -FLT_MAX;
    for (int i = 0; i < n; i++) {
        // 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;
        }
    }
}
```

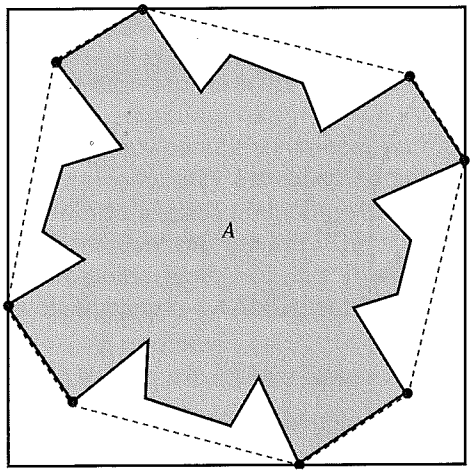

Figure 4.6 When computing a tight AABB, only the highlighted vertices that lie on the convex hull of the object must be considered.

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).

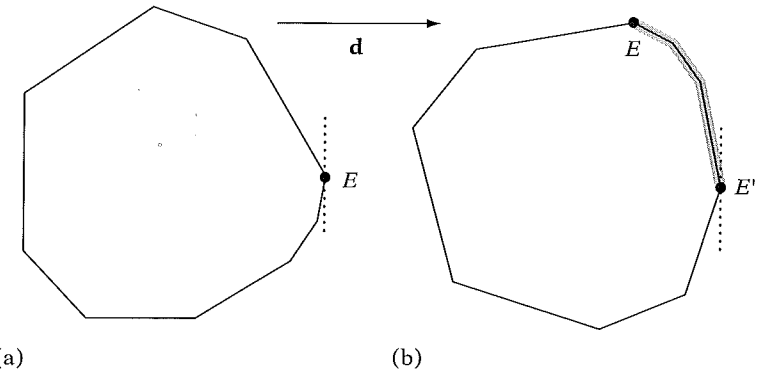

Figure 4.7 (a) The extreme vertex E in direction **d**. (b) After object rotates counterclockwise, the new extreme vertex E' in direction **d** can be obtained by hill climbing along the vertex path highlighted in gray.

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 $\mathbf{M}$, 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 $\mathbf{M}$ 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 $\mathbf{M}$ are the axes. If instead the vectors are multiplied on the left of the matrix as row vectors, then the rows of $\mathbf{M}$ are the axes.)

Say A is given using min-max representation and $\mathbf{M}$ 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 $\mathbf{M}$ contribute. Therefore, finding the extents corresponds to finding the vertices that produce the minimal and maximal products with the rows of $\mathbf{M}$. 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];
```

Computing an encompassing bounding box for a rotated AABB using the 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 (int i = 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 (int j = 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 (int i = 0; i < 3; i++) {
        b.c[i] = t[i];
        b.r[i] = 0.0f;
        for (int j = 0; j < 3; j++) {
            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 freely oriented bounding box. Oriented bounding boxes 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 (int i = 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 (int i = 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

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