Notes on Real Time Collision Detection

Mauricio Andres Rovira Galvez

March 7, 2016

Contents

List of Algorithms

Listings

Chapter 1

Introduction

The topic of collision detection is of particular importance in several branches within computer graphics. Some examples include:

- It is used in video games to create realistic animations and interactions.
- It is used to render a frame in a movie.
- It is used in general animations.
- It is used in physical simulations.

As a result, a lot of study has been devoted to this area. In these notes we will discuss two main components: collision detection and space partitioning algorithms.

Before we begin, it is important to determine what we would want in a collision detection system, as well as which are the challenges that we will face when designing it.

1.1 Design Factors

The first thing to consider is how objects are being represented. For the most part, they are represented as triangular meshes (since this is what graphics hardware is designed to work on). That being said, there are other ways of representing objects, such as modelling with implicit functions. While the choice of representation does affect the way the collisions (and other algorithms) will work, let us agree to only consider triangular meshes for the remainder of this discussion.

The second point of discussion is what we are going to be colliding. To be more concise, do we use the rendering mesh or something else? Let's look at an example: suppose that we have a mesh that is composed by 2,000,000 triangles. We wish to intersect this mesh against another model that is also composed of 2,000,000 triangles. The näive way of performing this task would be to check every triangle in the first mesh against every triangle in the second. This, however, results in 4,000,000,000,000 intersection tests! Even with the most powerful computers available, this would take a while. Even worse, it this were part of a game, the amount of time the player would have to wait *per* frame is unacceptable.

Clearly in this case we would want to use something *other* than the render mesh, preferably something that is more compact. This is where bounding volumes kick in. These volumes are called proxy geometry. These are usually optimized for collision detection systems. Unfortunately, they

also represent a problem: suppose that all of the proxies are computed in a pre-processing step. What happens when the original mesh is altered? How do we maintain these new geometries?

Chapter 2

Bounding Volumes

We begin our discussion of bounding volumes by asking the following question: what kind of properties do we want in a bounding volume? Ideally, we would want the following:

- Inexpensive intersection tests.
- Tight fitting.
- Inexpensive to compute.
- Easy to transform (rotations are of particular importance).
- Use little memory.

Unfortunately as we will soon see, there have to be trade-offs whenever we select a bounding volume. For example: a sphere is trivial to test and occupies little memory (it only requires 4 floats). Conversely, a convex hull gives us a very tight bound on the object, but requires a lot more memory and is more expensive to test for intersections. So the choice of bounding volume ultimately boils down to the specific requirements of our application. In this chapter we will be focusing on the two most common examples of bounding volumes: axis-aligned bounding boxes (AABB), and spheres. At the end we will briefly discuss another volume that has become prevalent in the industry: sphere-swept volumes.

2.1 Axis-aligned Bounding Boxes

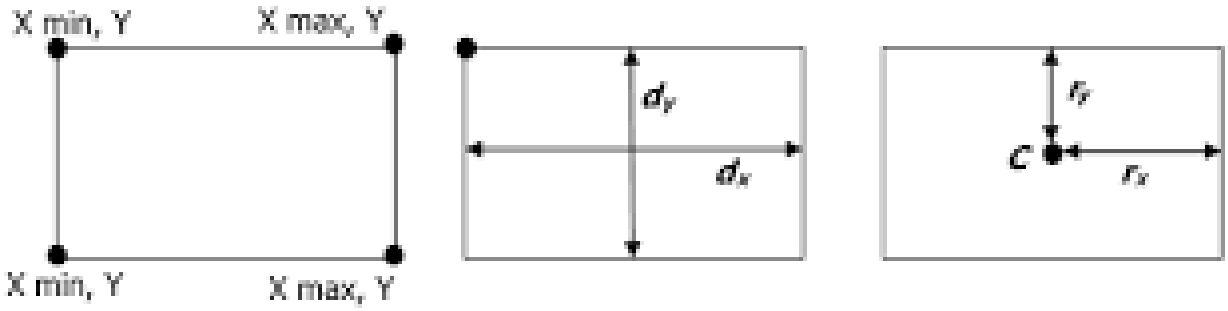
This is one of the most common bounding volumes used in the graphics industry. We define an axis-aligned bounding box (or AABB for short) as follows:

Definition 2.1.1 (Axis-aligned Bounding Box)**.** An *axis-aligned bounding box* is a rectangular six-sided box that is characterized by having the normals of each face parallel with the axes of the given coordinate system.

The property of having the normals remain parallel with the coordinate axes is what gives the AABB such a fast intersection test, since we only have to compare each individual coordinate value directly.

The three most common ways to represent an AABB are as follows:

Figure 2.1: The three representations of an AABB (from left to right): min-max, min-widths, and centre-radius



1. **min-max:** This form stores two points, representing the minimum and maximum extents of the box. In code, this would be:

Listing 2.1: min-max AABB

```
struct AABB
{
  Point min;
  Point max;
};
```

2. **min-widths:** stores the minimum corner of the box, along with its 3 dimensions (width, height, and length). In code, this is:

Listing 2.2: min-widths AABB

```
struct AABB
{
  Point min;
  float d[3];
};
```

3. **centre-radius:** stores the centre of the box and the radii in each of the 3 axes. In code, this is:

Listing 2.3: centre-radius AABB

```
struct AABB
{
  Point c;
  float r[3];
};
```

The three different representations are also shown in Figure 2.1.

So which one of these ways is more efficient in terms of storage? At first glance, it may seem tempting to say that they are all equivalent. After all, we ultimately have to store 6 floats (or

doubles) in all three. While this may be the case, there are some optimizations that can be done. For example, the dimensions of both the centre-radius and the min-widths can be stored using less bits than would normally be required. Also, if we set things correctly, it is possible to store the dimensions (or radii) as integers, which can be more tightly packed than floats. With this in mind, it is clear that the worst representation is the min-max, since we are forced to always store 6 floats (or doubles) no matter what we do.

In terms of updating, consider the following:

Example 2.1.1. Consider a sequence of n rotations and m translations, each one represented by a 3x3 (or 4x4 if we are working in homogeneous coordinates) matrices. Suppose that each transformation occurs at each time step of our simulation. Which AABB representation requires the least possible number of operations?

For each possible transformation, we incur in a multiplication of a matrix by a vector. Since this is constant, then the only way of optimizing the number of operations is to reduce it. Looking at the three representations, both min-widths and centre-radius require one multiplication, while min-max requires two.

At this point it would be fair to ask why we would want to use the min-max representation of the AABB at all. The reason becomes apparent when looking at intersection tests and updates as we will discuss next.

The intersection test for AABBs is straight-forward: 2 AABBs overlap if they overlap on all axes. Regardless of the representation, the algorithm is the following:

Algorithm 2.1 AABB intersection test for two boxes A and B

```
r ← 1
for each coordinate axis do
    if A_max < B_min or A_min > B_max then
        r ← 0
    end if
end for
return r
```

The optimizations that can be done here mostly depend on the implementation that is being used and what functions are available. Based on the algorithm presented in 2.1, we can see that both the min-widths and centre-radius representations are less optimal than the min-max, since the ranges need to be computed manually before comparison, as opposed to the min-max that can compare directly.

Now we turn our attention to updating AABBs. Clearly translations preserve the alignment of the boxes, but rotations present a particular problem, as we will see next.

2.2 Computing and Updating AABBs

Recall that AABBs have their face normals parallel to the *given* coordinate axes. This means that the AABB is bound to whatever coordinate system we used to create it. More specifically, it is bound to the space where the box was constructed. Generally, this space is the model space for the particular object we are bounding. This leads us to an interesting problem: how do we intersect two boxes that are defined in different spaces?

Let us examine this question with the following example:

Example 2.2.1. Consider two boxes defined as shown in Figure **??**. This figure shows the boxes in world space coordinates. We have two possible choices of space to transform the boxes to check for intersection: A or B. In which do the boxes intersect?

From the diagram it is clear that the boxes are barely touching in world space coordinates. Suppose we transform them into the space of A. After all the translations and rotations take place, we can see that B would intersect with A. Now let's look at B. Notice how A no longer intersects?

This example shows the reason why the choice of space is important for performing intersection tests. It also reveals a more subtle problem. Whenever we transform from one object space into another, we incur in matrix multiplication(s). Since all operations are done in floating point, then these transformations introduce additional errors to our computations. More importantly, if we transform to a space that is further away from the origin, such as B, then the values required to perform the operation are larger, and so the errors scale. Conversely, if we take a space closer to the origin, such as A, then the values are smaller and the error decreases.

So the important point here is the following: whenever we choose the space to compute intersections, care must be taken to ensure that it introduces the least error possible, while keeping in mind that it may give false results. With this in mind, lets now look at how to update bounding boxes after transformations (specifically rotations) have occurred.

Since AABBs have to preserve the alignment to the axes (otherwise we loose the fast intersection tests), then special care must be taken when they are rotated. Whenever we rotate an AABB, it must be updated to preserve its alignment to the axes. The most common strategies for this are:

- Use a loose-fitting AABB that always encloses the object.
- Compute a tight dynamic reconstruction from the original set.
- Computing a tight dynamic reconstruction using hill climbing.
- Computing an approximate dynamic reconstruction from the rotated AABB.

Let us examine each one in a bit more detail.

The first option exploits the fact that spheres are invariant under rotations. So the strategy is the following: construct an AABB that bounds the bounding sphere of the object. The sphere is centred at the pivot for the object, and the radius is set to be the point that is furthest away from the pivot. This guarantees that no matter what rotation occurs, the object will always remain inside the box and no changes need to be made. There are two points of consideration here: first off, the AABB is not tight around the object, which can lead to false intersections. The second point is the following: since a bounding sphere must be computed in order to determine the AABB, why not use the bounding sphere instead?
On the other hand, big advantage of this method is that unless the pivot for the object is changed, then the AABB does not need to be updated (save for translations).

The second option involves reconstructing the AABB from the points. This means the algorithm shown in 2.2.

Clearly this algorithm runs in $O(n)$. So how can we optimize it?
Well the only option available to us would be to remove as many points as possible from the original set. This can be done by noticing that only those points that sit on the convex hull of the object

Algorithm 2.2 Find the min and max points along an axis **d**

```
i_min ← −1
i_max ← −1
p_max ← ∞
p_min ← −∞
for i ≥ n do
    p ← pt[i] · d
    if p < p_min then
        p_min ← p
        i_min ← i
    end if
    if p > p_max then
        p_max ← p
        i_max ← i
    end if
end for
```

contribute to the size of the bounding box. That means that we can reduce the runtime of this from $O(n)$ to $O(k)$ where k is the size of the convex hull. This has two drawbacks: first, note that $k \leq n$, so in the worst case we would still have to traverse the entire set. Second, while it is possible to optimize the construction of the convex hull by using space partitioning techniques, this time is still used at pre-processing. At this stage we have already computed the convex hull, so we might as well use it. The only reason why we might care for the AABB containing the convex hull of the object is usually precision, at which point we should probably consider using the convex hull directly.
That being said, intersections against convex hulls are expensive to compute, so it could be worthwhile to wrap the convex hull inside an AABB to optimize this.

The next case is easily shown with the following example.

Example 2.2.2. Consider a regular n-gon (like a hexagon for example) that has a vertex v with coordinates $(x, 0)$ where $x > 0$. Clearly this point is the maximum on the x-axis. Now suppose we rotate the polygon counter-clockwise by some angle θ. Which point is more likely to be the new maximum?

In this example, we can clearly see that we can ignore vertices on the opposite side of the polygon (provided θ isn't too large, as we will discuss later). This means that the only vertices that we have to look at are those that are neighbours of v, which cuts down on our search dramatically. This is called hill-climbing.

This technique requires two things: first off, we need an efficient way of obtaining the neighbours of a given vertex. The second requirement is that the object must be convex (or we can operate directly on the convex hull of the object). Its advantage is clear: it reduces the cost of updating the AABB while at the same time providing a very tight fit for the particular object. It does however suffer some drawbacks. First, it is perfectly possible to encounter a situation where the search through the neighbouring vertices does not yield the real maximum (or minimum) in that direction. This means that there needs to be some pre-processing done for the object before the update is called, or the update function must be robust enough to handle these cases.

The second issue comes from the size of the neighbourhood. If the rotation is relatively small, then we have to search through a relatively small number of vertices. Unfortunately if the object is spinning too quickly, then the size of the neighbourhood must increase to accordingly, which can lead to lower performance.

The last of the four realignment methods is to simply wrap the rotated AABB in a new AABB. This gives us a new (approximate) AABB that we can use in our computations. The first point of consideration is the following:

Case 1. Consider an object with a bounding box A. Now suppose that we apply a sequence of rotations $R_1, R_2, \ldots, R_n$. Suppose that for each rotation, we wrap the rotated AABB with a new AABB. In other words, A_1 is constructed from A, A_2 from A_1 and so forth. What happens to the box? How can we address this problem?

This sequence of operations results in the box growing indefinitely. To solve this, we simply construct the new AABB based on the *original* AABB. Since all the transformations can be easily accumulated into a single matrix, we simply take the original box, transform it to whatever step we are currently at, and use that transformed box to compute the new one.

The general idea for the algorithm is the following: consider a box A that was transformed by a matrix $\mathbf{M}$ resulting in a box A'. First of all, recall that a matrix is ultimately a transformation of the basis (the coordinate axes) of that vector space (the space where the box was defined). That means that encoded inside the matrix $\mathbf{M}$ are the new coordinate axes that we are mapping to. The next thing to consider is this: if we rotate the minimum and maximum extents of the box, then the new maximum and minimum extents will be composed by a linear combination of the originals. This all implies that we can simply examine each coordinate individually by looking at the appropriate column (or row) on the matrix $\mathbf{M}$ and the transformed values of the minimum and maximum of A.

To better understand this, consider the following example:

Example 2.2.3. Consider a box A that has maximum $(1, 1, 1)$ and minimum $(-1, -1, -1)$. Suppose we apply the following rotation matrix:

$$\mathbf{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

Then the new maximum on the x-axis can be computed as follows:

$$\begin{aligned} x'_{\max} = &\max(\mathbf{R}_{0,0}x_{\min}, \mathbf{R}_{0,0}x_{\max}) \\ &+ \max(\mathbf{R}_{0,1}x_{\min}, \mathbf{R}_{0,1}x_{\max}) \\ &+ \max(\mathbf{R}_{0,2}x_{\min}, \mathbf{R}_{0,2}x_{\max}) \end{aligned}$$

As we can clearly see, we only need to consider the first column of the matrix and the x-coordinates of the minimum and maximum to compute the new x maximum of the box.

Next, we turn our attention to bounding spheres. As we will see, the interesting part about spheres isn't so much how they are represented or updated, but rather how they are created in the first place.

2.3 Spheres

Spheres are another very common bounding volume and serve as the foundation for another volume found throughout the industry. As we might expect, their intersection tests are very straightforward as is their representation. To represent a sphere, we just need a centre and radius.

Listing 2.4: Sphere representation

```
struct Sphere
{
  Point c;
  float r;
};
```

The intersection test is also pretty simple as is shown in 2.3.

Algorithm 2.3 Sphere intersection test of two spheres S_1 and S_2

$\mathbf{d} \leftarrow \mathbf{c}_1 - \mathbf{c}_2$
$dist \leftarrow \mathbf{d} \cdot \mathbf{d}$
$sum \leftarrow r_1 - r_2$
return $dist \leq sum \cdot sum$

In addition to this, spheres are invariant under rotations, which means that their update methods are trivial: we just need to translate the centre as is required.
At this stage is worth asking the following: if spheres are so easy to use, why aren't they used all the time? Why bother with other volumes? The answer is two-fold: first, while spheres are very convenient, they don't always provide the best bounding for an object. Second, how do we compute a sphere that is tight around the object?

We will focus on answering the second question for the remainder of the chapter. The first question mostly depends on the application and what the requirements are, so it will be omitted.
A first attempt would be to try and find the centre of the mesh (finding the geometric mean of all the vertices) and use this as the centre of our sphere. The radius would then be the largest distance from this point. As attractive as this idea may seem, it actually leads to very bad bounding spheres in the case where the points are fairly spread out (up to twice the needed radius, in fact). So how can we improve this?
Instead of taking the geometric mean, we could construct an AABB around the points and then use its centre as the centre of the sphere (it is actually the opposite of trying to find an invariant AABB).

An alternative way of computing the bounding sphere is to find an approximate bounding sphere (one that doesn't necessarily bound all of the points) and then refine it. The idea is the following: find 6 points (not necessarily unique) and select the pair that is furthest apart. The centre of the sphere becomes the halfway point between these two and the radius is the half distance between them. The next step of the process is to iteratively loop over all the points on the mesh and check if they are already bound by the sphere. If they aren't then the sphere is created to encompass the old sphere and the new point. This is done by taking the diameter of the sphere to be the distance between the new point and the point that sits on the old sphere that is opposite to the new point with respect to the old sphere centre.

The idea of this algorithm will be the theme for the rest of the chapter. Essentially we start with a guess (how good this guess is depends on the method) and then we iteratively refine it until we get a sphere that bounds the entire object.