Bounding Volumes & Bounding Volume Hierarchies

Suppose that we are tasked with designing a ray tracing system for a high-end movie studio. We are provided a scene that consists of several meshes. The smallest meshes consist of approximately 500 triangles, while the more complex meshes would be in the order of millions of triangles. With our current implementations, the ray tracer would have to test *every* triangle for intersection in order to determine what to shade. This would be done for every single sample and for every single pixel. Even with techniques such as multithreading, the render times would still sky-rocket due to the sheer number of intersection tests. Moreover, most of these intersections would be completely unnecessary.

Now let's look at a different problem. Suppose we are designing a video game. In the game, we need our character to be able to collect objects, and determine when the character has been hit by enemies. All of these tasks will be handled by a system called collision detection, which basically determines whether an object is in contact with another. In modern 3D video games, character models tend to have tens of thousands of triangles. Trying to determine whether a character is in contact with an object would require us to test *every* single triangle. In an environment where we have 16ms to present the next frame, this would be completely unacceptable, especially since most of the intersection tests would be unnecessary like in our ray tracing example.

What both of these situations have in common is the need to reduce the number of intersection tests when dealing with very large and complex objects. What we would ideally want is a way of avoiding the need to compute intersections until we were absolutely sure that we had to do them. Even then, if we know that we have to compute an intersection, we would like to be able to compute it only for the geometries that are most likely to be hit given the region we are interested in.

There are several possible solutions to these problems, but most of them rely on somehow subdividing the space so that we can more easily discard regions that we are not interested in. One of these techniques is called the bounding volume.

Bounding Volumes

The idea behind the bounding volume is the following: we have a very complex object for which computing intersections would take a long time. What if instead of intersecting against that complex object, we created a simpler object as a proxy and we intersect against that? So, instead of intersecting against a mesh consisting of millions of triangles, we could instead intersect against a much simpler object that would act as a proxy for the mesh. Since computing intersections against this proxy would be a lot faster, it wouldn't matter if we are testing it every time. If we don't hit the proxy, we know there's no way the mesh will be hit. If the proxy is intersected, then we can compute the intersection

against the mesh itself, since we know that it will be hit.

These simpler proxies are called *bounding volumes*, and here are some of the things we would like for them to be:

- Inexpensive intersection: we need the volume to be easier and faster to intersect than the original object, otherwise there is no reason to have it in the first place.
- Tight fitting: what the volume is ultimately doing is wrapping the object. If the wrapping is very loose, there will be more areas where the volume will be intersected, but the mesh will not. Ideally we want it to be as tightly wrapped as possible.
- Inexpensive to compute: this is especially important for real-time systems. If we need to update or change the volume because the underlying object is changing, we want this to be done as quickly as possible.
- Easy to transform: as the underlying object changes, we need to be able to easily apply the same transform to the volume and have it retain its properties.
- Use little memory: similar to the intersection requirement, we need the volume to not take up a lot of space, otherwise we are left with two memory-expensive objects to deal with.

As with most things in computer science, there isn't one solution that satisfies all of these requirements. In general, the tighter the volume becomes on the object, the more expensive everything else gets. Below is a list of several types of bounding volumes along with a short analysis of each one. The items are ordered with the faster, less memory intensive options going first, and the better bound options going last.

1. **Bounding sphere:** take your object and wrap it in a sphere. The main advantage of this approach is that the sphere requires very low memory (only 3 floats for the centre and one for the radius), has trivial intersections, and is completely invariant under rotations. The downside is that updating the sphere is non-trivial, and so is computing the initial sphere. The other problem is that it is not tightly fit around the object, causing a high number of false-positives.
2. **Axis-Aligned Bounding Box (AABB):** wrap your object with a box whose faces are aligned with the 3 canonical axes. There are several ways of representing it:
    - We can use two points which are the top-left and bottom-right of the box (or bottom-left and top-right, makes no difference here). This would require a total of 6 floats.
    - We could have a single vertex along with the width, height, and depth of the box. This would require 6 floats as well.
    - We could have a single point representing the centre of the box, and then the half-distances for the width, height, and depth of the box. Would also require 6 floats. This last approach in particular can be compressed to use less memory if we assume that all of the numbers

are integers, as we can pack multiple values in a single integer.
The intersection against a bounding box is trivial, but the box does not react well under all transformations. Specifically, if we rotate an AABB, it will loose its axis-aligned property, which is the reason why intersections are so fast in the first place. Therefore, any update that invalidates the AA property will require the box to be re-computed. In terms of

3. **Object Bounding Box (OBB):** The same as an AABB, but without the axis-aligned property. Intersections are not as trivial but still pretty fast, and the amount of memory is roughly equivalent to the AABB. Note that any OBB can be easily turned into an AABB. There are several ways of representing it, most commonly with 8 vertices or 24 floats.
4. **Convex Hull:** take your object and use a polygon to trace the outline of the object. This is called a *convex hull*. Out of all the volumes here it is by far the most tight-fitting, but the it is expensive in memory and to compute intersections against. Any changes to the underlying object require the hull to be updated as well.

These volumes are summarized in the table below:

Volume	**Memory Cost**	**Intersection Cost**	**Update Cost**
Sphere	4 floats	Trivial	Non-trivial to compute initially, invariant under rotations
AABB	6 floats	Trivial	Easy to compute, needs to be updated when rotated
OBB	24 floats	Not as easy as ABB	Hard to compute initially and update
Convex Hull	$3n$ floats for n vertices	Hard	Hard

So, with the bounding volumes we have a way of having a proxy for an object that would be potentially easier to intersect against. Going back to the problem of a ray tracer, what we would do is wrap each object in a volume (say an AABB) and intersect against that. If the ray hits the box, then we can check to see which triangle it intersects against. However, what if the object is composed of multiple parts? Well, we could simply wrap each part into its own AABB, but if the object is composed of a lot of parts, then we would end up with a high number of AABBs. While the intersection test is cheap, it will grow linearly with the number of objects that conform the mesh. What can we do here?

A similar problem arises with the collision detection. Yes, we can wrap each

individual part of the character in its own AABB, but we end up with a similar problem to ray tracing. We can propose the following idea: if we wrapped the components of the models into volumes to act as proxies, then why don't we wrap the volumes themselves into other volumes to act as proxies for those? We can continue this until we have a single volume encompassing the entire model, which itself would be composed of smaller volumes until we hit the mesh itself.

Bounding Volume Hierarchy (BVH)

What we have just constructed here is a *bounding volume hierarchy* (BVH). These are very common in graphics as a way of performing two tasks at once:

- First, we once more delay the intersection tests by ensuring that the region that we are interested in actually contains a part of the mesh.
- It divides space into regions, which can then be culled so we can focus only on the relevant portions.

The questions that usually come up with BVHs are:

- How do we make them?
- How do we decide their depth?

As with most things in graphics, the solutions to these problems is very much dependent on the problem we have. That being said, we will discuss three strategies for creating a BVH:

- **Top-Down:** start with a single volume that encompasses all objects. Then, divide this volume into portions of related objects (can be based on their positions in space or if they are connected by another property). Continue dividing until each object has its own volume. This defines the BVH.
- **Bottom-Up:** start with each object wrapped in its own volume. Next, begin wrapping clusters of objects that are related (either by distance or some other property) in their own volumes. Continue this process until you are left with a single volume containing all objects, which defines the BVH.
- **Insertion:** Start with an empty BVH, and add the first object. Now add the next object and wrap them both in a volume. Continue adding objects. If they are unrelated to each other, add them as children of the root volume. Stop when all objects are contained in the BVH.

The bottom up is simpler to implement of all of the objects already have their own volumes, but may not yield the best tree for the application.

To intersect against a BVH, the process is the following:

1. Intersect against the root of the BVH. If there is a hit, then proceed. Otherwise there will be no hit against any members of the BVH.
2. If there was a hit, then intersect against every volume that is a child of the root. If there are no hits here, stop. If there is a hit, store that node.

3. For every node that was hit, intersect against their children. Repeat this process. If any hits make all their way down the tree to the leaves (which would be the objects), then perform the intersection against the objects themselves.

BVHs are a type of *space partitioning* structure that allow us to divide the region were are interested in into sub-regions, and therefore allow us to quickly cull out parts that are not relevant to the current operation. The main advantage of the BVH is that it adapts more easily to the existing geometries (especially if the volumes it consists of are relatively tight), but it does have some problems. Specifically, the intersections may not always be as easy, it may require more memory depending on the volumes it contains, and it is hard to navigate, especially if we are starting somewhere inside the hierarchy itself. Next time, we will discuss regular grids.