

# Computer Graphics – Lab Assignment

## Ray-Tracing 2

November 2, 2010



## 1 Introduction

After last week's introduction to the basics of ray-tracing we're now going to improve the rendered images and make it possible to render 3D models consisting of thousands of triangles.

## 2 Speeding up intersection testing

As you know, the basic operation from which ray-tracing derives its name is intersecting a ray with all the 3D objects in the scene and finding the intersection closest to the ray origin. If you were to do measurements on last week's framework you would see that the code spends an awful amount of time calculating intersections of rays with triangles, up to 80% of the rendering time. And the 3D scene we used so far was extremely simple, consisting of only 25 triangles (and 3 spheres).

The fundamental problem of the code used so far is that for every ray that is shot it tests that ray for an intersection against *every* triangle in the scene. More formally, the intersection algorithm's execution time is of order  $O(n)$  in the number of triangles, meaning that it performs linearly; if we increase the

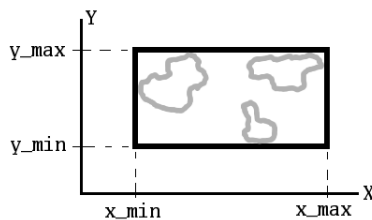


Figure 1: A (2D) axis-aligned bounding box, with the 3 objects it encloses

number of scene triangles  $n$  by 10, testing a ray against the scene will take 10 times as much time. Clearly, we want to be able to use more interesting models consisting of thousands of triangles, but not at a cost of hours of rendering time.

In this week's assignment we are going to use a *Bounding Volume Hierarchy* (or BVH) to improve on this, giving us a much better  $O(\log n)$  performance. For large  $n$ , this makes a huge difference. Note that the spheres in a scene are not handled using the BVH, as there are usually only a few of them.

The basic idea of a BVH is that small groups of triangles (located close together) are enclosed in a bounding volume, in this case an *axis-aligned bounding box* (or AABB for short). An AABB is basically a box-shaped region whose sides are aligned with the axes of the coordinate system, see Figure 1. Then, if we need to test a ray against the scene for intersections we can use the fact that if that ray doesn't intersect a bounding box it will also not intersect any of the triangles enclosed by that bounding box. If a ray does intersect the box, however, it might intersect one of the triangles inside it.

Enclosing small groups of triangles with a bounding box is a start, but it will merely replace thousands of triangles with hundreds of bounding boxes containing a handful of triangles each. We still have to test a ray against all of the boxes.

Going one step further we pick two bounding boxes that lie close together and compute a new bounding box that encloses that pair. We can continue this process until we are left with one top-level bounding box that will enclose the whole scene. By keeping track of which bounding box encloses which two others we can create a tree of bounding boxes, a BVH.

The nodes of the BVH are either *inner* nodes or *leaf* nodes. Inner nodes always have two child nodes, pointers to which are stored in the inner node. The second type of node, the leaf node, stores a list of triangles.

Both types of nodes store a bounding box. For inner nodes this bounding box is guaranteed to surround the bounding boxes of its children. The bounding box of a leaf node is guaranteed to surround all triangles stored in that node. See Figure 2 (note that the figure shows the nodes in 2D instead of 3D, for illustration purposes).

Intersection of a ray with a BVH is quite straightforward. We first test if the ray intersects the root node of the BVH, using the root node's bounding box.

If this bounding box is intersected we check for each of the root's child nodes if the ray intersects that child's bounding box. We traverse down to the child nodes that are intersected and continue testing for intersections and traversing down until we reach a leaf node. When we reach a leaf node we check the ray against all triangles stored in that leaf node. During the traversal of the BVH we keep track of the triangle intersection closest to the ray origin found so far.

The intersection speed-up a BVH provides comes from the fact that during traversal we can skip child nodes whose bounding box is not intersected by the ray being tested. Not only will we skip that single child node, but implicitly also the whole subtree below it, including the triangles in the leaf nodes. The subtree skipped this way can contain large parts of the scene.

## 2.1 Framework

Writing code to build a BVH for the triangles in a 3D scene is quite a bit of work and would not be possible in the time allocated for the lab session, so the framework already contains this code. If you're interested in the build process, Appendix A contains a description.

The framework will build a BVH for the loaded scene and has functionality to view the tree nodes, so you can get an idea of how the BVH is organized. When you load, for example, **buddha.scn** and press the **B** key (uppercase) you will see lots of blue boxes drawn in the scene. These are the bounding boxes of the leaf nodes. Remember that each leaf node contains a small group of triangles.

When you press the **]** key once the blue boxes disappear and a red box appears. This red box is the root node's bounding box. When you press **]** again you step one level deeper into the BVH and the bounding boxes of the inner nodes at that level will be shown. With **[** you go one level higher up in the tree.

## 2.2 Assignment

What is missing in the framework is a way to use the built BVH to quickly find groups of triangles that can possibly be intersected by a ray and test them for

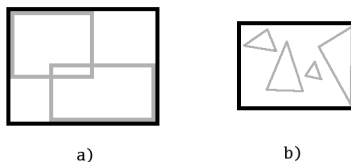


Figure 2: a) an inner node with its bounding box (the black rectangle) and the bounding boxes of its two child nodes; b) a leaf node's bounding box with the triangles it contains.

intersection. So, what is missing are the pieces for *traversing* the BVH for a given ray and finding the closest triangle intersection (if there is any).

Implement the function `find_first_intersected_bvh_triangle()` in *intersection.c*. To check for an intersection between a ray and a bounding box you can use `bbox_intersect()`, declared in *bbox.h*. For testing a ray against a triangle you can use `ray_intersects_triangle()` in *intersection.c*.

Test with *simple.scn* first, before trying *cow.scn* and *buddha.scn*. You can switch between using the BVH or the previous intersection code with the `b` key, to verify correctness of your BVH traversal code. This might not be feasible for the Buddha scene, due to the long rendering times of the non-BVH code for large numbers of triangles <sup>1</sup>.

**Note 1:** Because of the data type used to implement a BVH node (a C `union`) access to its type-specific fields is best done through the functions declared in *bvh.h* starting at `inner_node_left_child()`.

**Note 2:** The root node of the BVH is available through the pointer `bvh_root` declared in *bvh.h*.

**Note 3:** Remember that we’re trying to find the *first* triangle intersection for a ray. Once we have found a triangle intersection during BVH traversal we can easily reject BVH nodes that will not give us an intersection closer to the ray origin. A similar optimization is also possible for the case of deciding which of the two child nodes of an inner node to process first. *Be sure to implement both these optimizations.*

**Note 4:** The framework prints some statistics at the end of rendering the image. The average number of triangles tested for intersection per ray is useful to determine if your optimizations actually improve BVH traversal.

### 3 Anti-aliasing

If you look at the ray-traced images that the framework renders you might notice the hard (and ugly) outlines of objects, see Figure 3(a). There you can see the “staircase effect” on the edge where the ground plane meets the white background. Similarly, there is a hard edge between the small sphere and the ground plane in the back.

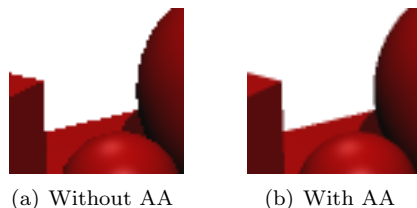


Figure 3: Image quality w.r.t anti-aliasing

---

<sup>1</sup>Which was, of course, the reason we introduced the BVH

This is due to the fact that we only shoot one ray per pixel, and each pixel therefore only shows the shading of at most one object. The ray shot through the pixel next to it might show another object and therefore another shading value, which might have a noticeably different color.

To overcome this we can shoot multiple rays per pixel and compute the average color of the colors returned for the rays. This form of *anti-aliasing* works reasonably well, as shown in Figure 3(b), as it decreases the color differences between neighboring pixels.

The framework currently allows you to toggle between anti-aliased and non-anti-aliased rendering with the `a` key, which sets or clears the **`do_antialiasing`** flag, but anti-aliased rendering is not implemented yet. Alter the function **`ray_trace()`** in *main.c* so that, when **`do_antialiasing`** is set, it shoots 4 rays per pixel and averages the resulting colors to get the final pixel color. Conceptually, each pixel should be divided into 4 sub-pixels (2x2), and a ray is shot through the center of each of the 4 sub-pixels.

Note that when you're viewing the ray-traced output and you toggle anti-aliasing the framework will immediately re-render the image.

## 4 Grading

A fully correct implementation of the traversal of the BVH which finds the closest intersected triangle is worth 7 points.

If your implementation doesn't always correctly find the the first intersection you can loose up to 2 points. When you don't include the optimizations described in Section 2.2 you can also loose 2 points.

Correct implementation of anti-aliasing is worth up to 2 points.

You can get one additional point for writing clean, well-structured and well-commented code (on the opposite side, unreadable or overly complex code might cost you a point).

## A BVH construction

Here, we shortly describe the procedure used to build the BVH.

Starting with the triangles in the scene we compute the bounding box that tightly encloses these triangles, and compute along which axis the box is the longest. We then (conceptually only) place a plane perpendicular to this axis halfway in the box, dividing it in two equally sized halves. The triangles then get sorted into two groups, depending on which of the halves they overlap most. For each of the two groups we apply the procedure just described to each group independently.

At some point the number of triangles left in the current group becomes very small and it doesn't make much sense anymore to subdivide into two groups. In that case a leaf node is created which stores the triangles. In the framework,

a leaf node is created when no more than 4 triangles are left. The construction procedure also creates a leaf node when a maximum tree depth is reached.