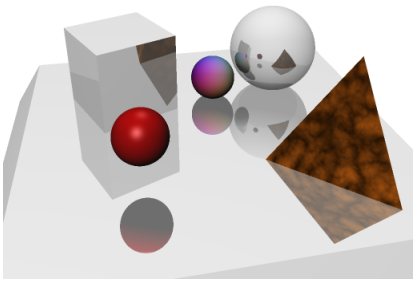
Graphics and Game Technology

Assignment 4

Ray-tracing



1 Introduction

Ray-tracing is a rendering technique that has been around since as early as the late 1970s. For years it has been a technique that was mostly used to create very realistic looking images, at the price of many hours of computing time. Until quite recently, even companies like Pixar (of "Finding Nemo" fame) used ray-tracing in their movies only if regular scan-line rendering could not create a certain effect, like complex reflections from glass objects. And even then, they used it for just a few shots.

Ray-tracing has regained interest in recent years since *real-time* ray-tracing has become possible on commodity computer systems. This is partly due to increasing CPU speed (Moore's law), but also the availability of multi-core/multi-CPU systems together with the fact that ray-tracing is relatively easy to parallelize.

In this assignment we are not going to focus on real-time ray-tracing (sorry), but you will get acquainted with the basics of the technique.

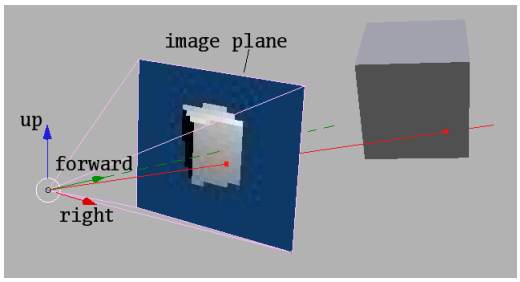


Figure 1: Camera in a 3D scene, with one camera ray.

2 Part 1 - Ray-tracing basics

As a reminder, here is a really short and high-level description of the ray-tracing process for creating an image. See also the sheets presented at the lectures and the chapter on ray-tracing in Shirley et. al.

See Figure 1: from the camera position, rays are shot through pixels in the image plane into the 3D scene. A ray starts at a certain 3D coordinate (its origin) and runs straight in a certain direction.

Each ray shot into the scene is tested for intersection with the objects in the scene. The first intersection that is found – the one closest to the ray origin – is used to determine the surface colour "seen" by the ray (this process is called "shading").

A shading routine (also called a "shader") uses values such as the surface normal at the point of intersection and the position of light sources to compute the colour at that surface point. The shader can also trace new rays through the scene to check, for example, from which light sources the surface receives light, or to create a reflective surface.

The final colour returned for a camera ray is the colour seen at that pixel of the image.

2.1 Framework overview

The framework is fairly elaborate, as it takes quite some code to get a simple ray-tracer.

A second file is also provided, `scenes.tgz`, that contains a number of 3D scenes that you can use to test your ray-tracer.

Two types of 3D objects are supported: spheres and triangles. We include spheres as they nicely show how ray-tracing can support "perfect" objects that can only be approximated using a large number of triangles when using scanline rendering.

The framework uses a very simple file format to describe 3D scenes. See the files with `.scn` extension for examples. The file format supports light sources (position, intensity), spheres (position, radius), object material definitions, and

triangle models from files with `.ply` extension (a number of these are also included). The main program expects the filename of a scene as its single argument.

Two view modes are available: an interactive OpenGL view, and a ray-traced view. As the ray-tracing process takes a bit of time to complete (usually a few seconds), the ray-traced view does not allow the viewpoint to be changed. You can switch between these two modes with the **r** key. A number of predefined viewpoints are available with keys **1** through **6**.

Important files are:

`main.c`	Main program
`types.h`	Definitions of several basic types used, such as a triangle, light source, an intersection point.
`scene.h/.c`	Contains the object in the 3D scene (lights, triangles, spheres, etc). Also contains scene specific values, such as camera position and image background colour.
`intersection.h/.c`	Ray-object intersection code
`shaders.h/.c`	Contains shading routines
`v3math.h/.c`	3D vector routines

3 Assignment 1: Camera rays

The first thing to add to the framework is code that shoots camera rays, i.e. rays shot from the camera position into pixels in the image plane. Each of these rays should go through the center of a pixel and determine the colour of that pixel.

Again, see Figure 1. There you see the camera coordinate system, in the form of the vectors **forward**, **up** and **right**, together with the camera's position in the 3D world.

In front of the camera is the image plane. The center of the plane is at a distance of one unit from the camera position in the **forward** direction. Look at function **ray_trace()** in `main.c`. This function already computes the image plane's size for you (which depends on the field-of-view of the camera) and contains an almost empty loop over all pixels.

Implement the missing pieces: for each pixel compute the point on the image plane that is the center of that pixel. Shoot a ray through that point and store the resulting colour in **color**.

The function **ray_color()** defined in `shaders.h` computes the colour "seen" by a given ray for you. This function works for camera rays as well as reflected and shadow rays. Its first parameter, *level*, will become clear further on in this assignment. Pass the value zero for now. Make sure you pass the correct values for the remaining parameters.

Test your implementation with the scene `silhouette.scn`. For this scene, only a very simple shader is used that simply always returns the colour red. This

means that when you correctly implement the computation of viewing rays, the ray-traced view should show a red silhouette of the scene, just like Figure 2(b).

Note: due to the fundamentally different methods of scan-line rendering and ray-tracing, the OpenGL view and ray-traced view may not match up pixel-perfect, as far as object boundaries is concerned. But differences of more than one or two pixels probably *are* a sign that something is wrong in the camera ray generation.

4 Assignment 2: Shading

4.1 Introduction

Now that we can trace camera rays we are ready to add some more interesting shaders so that we can render objects that look metallic, that reflect their surroundings, etc.

The fundamental operation for a shader is to compute the colour as seen by a ray at the point where that ray intersects an object's surface. To determine this colour, several values are available (see **struct intersection_point** in `types.h`):

p	The coordinates of the point to shade
n	Surface normal at the point to shade (normalized)
i	Direction *from* which the ray responsible for this intersection came (normalized). Note: his vector points *away* from the surface point!

Other values that are typically used during shading are listed below. Some of these are available directly in the code, others might need to be computed.

$\mathbf{l}_i$	Vector pointing towards light source i (normalized)
h	Vector halfway between **i** and $\mathbf{l}_i$ (normalized)
r	Direction of a reflected ray (normalized)
$\mathbf{c}_d$	A material's diffuse colour (its base colour)
$\mathbf{c}_s$	A material's specular colour (the colour of a highlight)
I_i	The intensity of the light coming from light source i
I_a	The intensity of the ambient light in the scene

4.2 A simple matte surface

To obtain more realism we are going to add a shader that varies the colour of a surface based on the amount of light reaching that point. The contribution of light can be computed using the dot product between the surface normal and the vector pointing to the light source, $\mathbf{n} \cdot \mathbf{l}_i$. Note that this dot product can be negative, depending on whether the light source is at the front or back side of the surface. In case the light source is at the back, there is no contribution from that light source.

We assume that all light sources in the scene emit pure white light, but they may vary in intensity. We also assume that all lights are "point light sources". See struct **light** in `types.h`. There is also a tiny amount of ambient light in the scene, which is also pure white, but with a very low intensity (defined by the variable **scene_ambient_light**).

Implement **shade_matte()** in `shaders.c` so that it returns a colour based on the total light contribution at the point to be shaded. Note that each of the three colour components must be in the range $[0, 1]$. Test the shader on the scene `matte.scn`. See Figure 2(c) for example output. Note: you don't need to use the material index anywhere. This is merely a field used internally by the framework.

4.2.1 Shadows

Next, we are going to add shadows to our matte shader. The insight here is this: if there is an object in the path of a ray from the point to be shaded to a light source, then there will be no light contribution from that light source.

Add the tracing of a shadow ray to the matte shader. Use the function **shadow_check()** defined in `intersection.h`. Again, test with `matte.scn`. See Figure 2(d) for example output.

You might notice that the spheres in the scene end up having black speckles. This is due to self-shadowing. Use a small offset in the ray origin to overcome this problem.

4.3 Blinn-Phong shading

The next shader you need to implement computes a surface colour based on three components: ambient, diffuse and specular intensities. The specular part gives metallic(-like) surfaces their characteristic look: they have a specular highlight where a bright spot is visible due to incoming light.

One formulation that allows these kinds of surfaces to be modeled is from Jim Blinn and Bui Tuong Phong. It determines the final surface colour $\mathbf{c}_f$ as follows:

$$\mathbf{c}_f = \mathbf{c}_d \left(I_a + k_d \sum_i \{ I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) \} \right) + \mathbf{c}_s k_s \sum_i \{ I_i (\mathbf{n} \cdot \mathbf{h})^\alpha \}$$

The constants k_d and k_s determine the contributions of the diffuse and specular components, so we can vary these relatively to each other. Note the parameter α in the last term. This value determines how shiny the surface will appear. The higher α is, the smaller the highlight will be.

Implement **shade_blinn_phong()** in `shaders.c`. Use $k_d = 0.8$, $k_s = 0.5$, $\alpha = 50$, $\mathbf{c}_d = (1, 0, 0)$ and $\mathbf{c}_s = (1, 1, 1)$ and include the tracing of shadow rays. Test with `blinn_phong.scn`. See Figure 2(e) for example output.

4.4 Reflections

Next, we are going to create a reflective shader. The trick here is to note that the visible colour of reflective surfaces depends on the colour of the surface itself combined with the colour reflected from the surface.

Suppose a ray intersects a reflective surface. To find the colour that is reflected, we simply shoot a ray from the intersection point in the direction in which the incoming ray is reflected. This direction depends on the surface normal at the intersection point and the direction of the incoming ray.

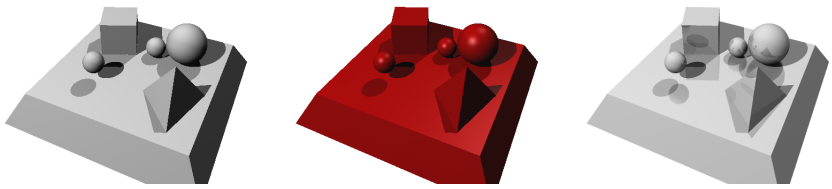
In terms of the values described in section 4.1 the reflected direction $\mathbf{r}$ is

$$\mathbf{r} = 2\mathbf{n}(\mathbf{i} \cdot \mathbf{n}) - \mathbf{i}$$

One possible problem is that rays might start bouncing around between reflective surfaces, as each intersection starts a new reflected ray. To overcome this problem there is a limit on the number of times a ray may be reflected, which is enforced by **ray_colour()**. Be sure to pass the correct value for *level* in your shader.

Implement **shade_reflection()** so that the surface colour that is returned consists of 75% matte shading and 25% reflected colour. The shader should also do shadowing. Test with `reflections.scn`. See Figure 2(f) for example output.

(a) OpenGL view (b) Silhouette shader (c) Matte shader



(d) Matte shader + shadows (e) Blinn-Phong shading (f) Mirror-like shading

Figure 2: (a) OpenGL view, (b)-(f) Ray-traced output for the different assignments

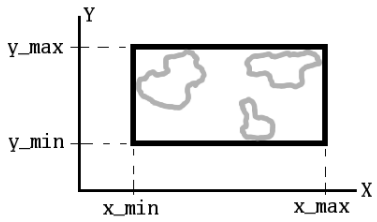


Figure 3: A (2D) axis-aligned bounding box, with the 3 objects it encloses.

5 Part 2 - Speeding up intersection testing

We are now going to improve the rendered images and make it possible to render 3D models that consist of thousands of triangles.

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 execution-profile measurements on the framework, you would see that the code spends a considerable 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, the ray is tested 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 number of scene triangles n by 10, testing a ray against the scene will take 10 times as long. Clearly, we want to be able to use more interesting models consisting of thousands of triangles, but not at a cost of hours of computing time.

To improve on this, we are going to use a *Bounding Volume Hierarchy* (or BVH), which will give 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 groups of triangles that are 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 3. Now we can use the fact that if a ray does not intersect a bounding box, it will also not intersect any of the triangles enclosed by it. Conversely: if a ray does intersect the bounding box, 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.

So we go one step further and 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 (see Figure 4).

The nodes of the BVH are either *inner* nodes or *leaf* nodes. Inner nodes always have two child nodes (i.e. pointers are stored in the inner node that point to the two child nodes). 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 4 (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 speed-up that results from the use of a BVH 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 do we skip that single child node, but implicitly also the whole subtree below it, including the triangles in the leaf nodes. The subtree that is skipped may contain large parts of the scene.

5.1 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 this assignment, so the framework already contains this code. If you are interested in the build process, Appendix A contains a description.

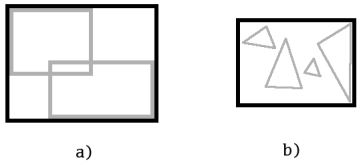


Figure 4: a) an inner node with its bounding box (the black rectangle) and the bounding boxes of its two child nodes; b) a leaf node's bounding box with the triangles it contains.

The framework will build a BVH for the loaded scene and has built-in 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.

5.2 Assignment

What is missing in the framework is a way to use the constructed BVH to quickly find groups of triangles that can possibly be intersected by a ray and to test them for intersection. In other words, 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 [1].

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 want 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.

[1] Which was, of course, the reason we introduced the BVH.

6 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 5(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.

(a) Without AA (b) With AA

Figure 5: Image quality with and without anti-aliasing.

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 5(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 are viewing the ray-traced output and you toggle anti-aliasing the framework will immediately re-render the image.

7 Grading

Correct computation of viewing rays can give you up to 3 points. A working matte shader (including shadows) can give you 3 points. Correct implementation of the Blinn-Phong and reflections shaders are worth 7 points each.

A fully correct implementation of the traversal of the BVH which finds the closest intersected triangle is worth 10 points. However, if your implementation does not always correctly find the the first intersection you lose up to 2 points. When you do not include the optimizations described in Section 5.2 you also lose up to 2 points.

Correct implementation of anti-aliasing is worth up to 6 points.

You can get up to 4 additional points for writing clean, well-structured and well-commented code (on the opposite side, unreadable or overly complex code might cost you 4 points).

Your final grade is the number of points divided by 4.

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 halfs. The triangles then get sorted into two groups, depending on which of the halfs 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.