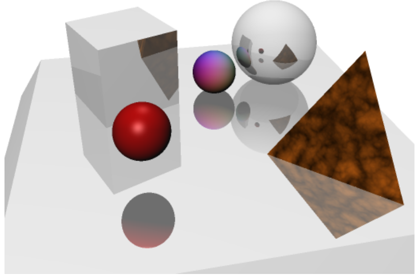
Computer Graphics – Lab Assignment
Ray-Tracing 1

October 11, 2010



Ray-tracing is a rendering technique that has been around since the late 1970's, but most people have always perceived it as "slow". For years it has been a technique that was mostly of use to create very realistic looking images, at a price of using hours and hours of computing time. Even companies like Pixar (of "Finding Nemo" fame) until quite recently only used it for just a few shots in their movies if regular scan-line rendering could not create a certain effect, like complex reflections from glass objects.

But ray-tracing has become of quite a lot of interest in recent years as the possibility of doing ray-tracing *in real-time* has become a reality. This is mostly due to Moore's law of increasing CPU speed, the availability of multi-core/multi-CPU systems these days together with the fact that ray-tracing is easy to parallelize.

In this week's and next week's assignment we're not going to focus on real-time ray-tracing (sorry), but you will get to discover the basics of the technique.

1 Introduction

1.1 A real quick reprise of ray-tracing

What follows is a really short and high-level description of the ray-tracing process for creating an image, as a reminder. Also see the sheets presented at the lectures and Shirley, Chapter 10.

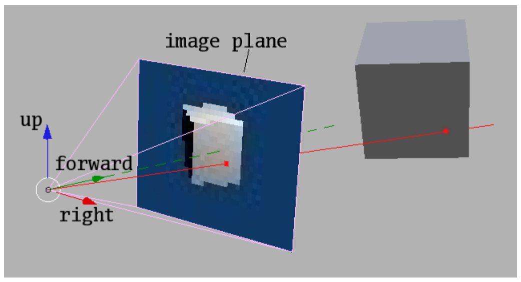


Figure 1: Camera in a 3D scene, with one camera ray

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

Each ray shot is tested for intersection with the objects in the scene. The first intersection found – the one closest to the ray origin – is used to determine the surface color "seen" by the ray (the process 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 color at that surface point. The shader can also trace new rays through the scene to check, for example, from which light sources the surface point to shade receives light, or to create a reflective surface.

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

1.2 Framework overview

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

There's also a second file provided, **scenes.tgz**, that contains a number of 3D scenes. These scenes are used for this week's and next week's assignment. This file should be unpacked in the same directory as where the framework is unpacked, don't unpack **scenes.tgz** within the framework directory!

There's two types of 3D objects supported: spheres and triangles. We included 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 scan-line rendering.

The framework uses a very simple file format to describe the 3D scene. See the files with extension *.scn* for examples. The file format allows lights to be specified (position, intensity), allows spheres to be specified (position, radius), allows the material for an object to be set and allows triangle models to be read from files with extension *.ply* (a number of these are included). The main program expects the name of a scene file as its single argument.

There are two view modes available: an interactive OpenGL view, and a ray-traced view showing the rendered image. 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, a light, an intersection point.
scene.h/.c	Contains the object in the 3D scene (lights, triangles, spheres, etc). Also contains some scene specific values, such as camera position and image background color.
intersection.h/.c	Ray-object intersection code
shaders.h/.c	Contains shading routines
v3math.h/.c	3D vector routines

2 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. Each of these rays should shoot through the center of a pixel and determine the color 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 color in **color**.

The function **ray_color()** defined in *shaders.h* computes the color "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 color 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 might 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.

3 Assignment 2 - Shading

3.1 Introduction

Now that we can trace camera rays we're going to add some more interesting shaders, so that we can render objects that look like metal, that reflect their surroundings, etc.

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

$\mathbf{p}$	The coordinates of the point to shade
$\mathbf{n}$	Surface normal at point to shade (normalized)
$\mathbf{i}$	Direction *from* which the ray responsible for this intersection came (normalized). This vector points *away* from the surface point!

Other values 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)
$\mathbf{h}$	Vector halfway between $\mathbf{i}$ and $\mathbf{l}_i$ (normalized)
$\mathbf{r}$	Direction of a reflected ray (normalized)
$\mathbf{c}_d$	A material's diffuse color (its base color)
$\mathbf{c}_s$	A material's specular color (the color 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

3.2 A simple matte surface

To get more realism we're going to add a shader that varies the color of a surface based on the amount of light reaching that point. The contribution of a 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 on the front or back side of the surface. In case the light source is on the back there's no contribution from that light.

We assume all light sources in the scene shine 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's 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 color based on the total light contribution at the point to shade. Note that each of the three color 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.

3.2.1 Shadows

Next, we're going to add shadows to our matte shader. The insight here is that if there is some object in the path of a ray from the point to shade to a light source then there will be no light contribution from that light source.

Add the tracing of 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.

3.3 Blinn-Phong shading

The next shader you need to write computes a surface color 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 due to incoming light is visible.

One formulation that allows these kinds of surfaces to be modeled is due to Jim Blinn and Bui Tuong Phong. It determines the final surface color $\mathbf{c}_f$ 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 set, the smaller the highlight.

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.

3.4 Reflections

Finally, we're going to create a reflective shader. The trick here is to note that the visible color of reflective surfaces depends on the color of the surface itself combined with the color reflected from the surface.

Suppose a ray intersects a reflective surface. To find the color being 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 3.1 the reflected direction $\mathbf{r}$ is

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

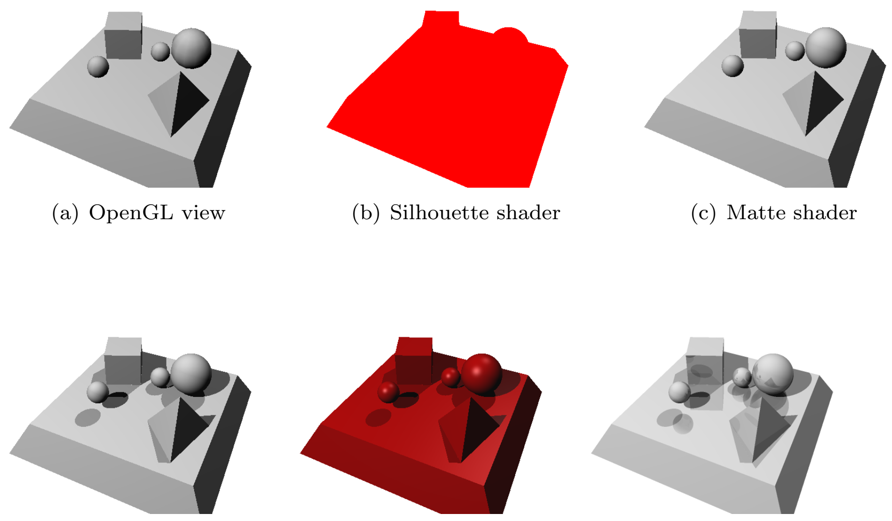
One possible problem is that rays might start bouncing around between reflective surfaces, as each intersection starts a new reflected ray. To overcome this problem there is a limit on the number of times a ray may be reflected, which is enforced by **ray_color()**. Be sure to pass the correct value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

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

4 Grading

Correct computation of viewing rays can give you 2 points. A working matte shader (including shadows) can give you 2 points. Correct implementation of the Blinn-Phong and reflections shaders are worth 2.5 points each.

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) 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