CSE 167 (Fall 2021) Raytracer Final Project Guide – II

This is the part 2 of the ray tracer final project guide, which is a replica of an assignment from previous CSE167s. The main difference from part 1 is that we provide test scene and reference output image for this part.

This assignment asks you to write a simple raytracer. Raytracers can produce some of the most impressive renderings with high quality shadows and reflections.

2.1 Introduction

You will be using a new scene format for this assignments. Scenes will be provided in txt format, and a skeleton scene parser `file.cpp` will be provided.

Unlike prior assignments, this assignment does not use OpenGL or hardware rendering at all, and will directly be written in software in C++. As such, all standard debugging tools are available (more straightforward than debugging shaders).

2.1.1 Logistics

In addition to the lecture material, online materials (see footnote)[1][2] are also great references. By starting with reading them, you may have a general idea the structure that is required in your code.

Traditionally, this assignment does not have a starter code. In this project, however, we provide a basic starter code for scene parser and file writer. (You may find it helpful to use some previous HW as framework.) Your task is to build a real system. For an example of some of the effects you can obtain, you could download public domain raytracers like PBRT or POVray. There is no objection to looking at the source code in them for inspiration and understanding, but all code you write must be your own. You are of course welcome to reuse code from previous assignments.

This assignment does not require (nor recommend) use of OpenGL. However, you may optionally use GLM functions for matrix-vector operations if you like. The next page briefly discusses how to output images with FreeImage. If you are interested in more details, you may also be interested in links to FreeImage Documentation[3].

2.1.2 Test Scenes

We do provide some test scene files. They are provided with the parser skeleton in `rt-testscenes.zip`, which has test scenes that are documented and have multiple camera positions you should try. There are also images of these scenes with different camera specifications. Note that these images were created in an OpenGL previewer and are a useful guide, but do not have the sophisticated shading, shadows and reflections, that your raytracer will provide for the same images.

2.1.3 Overview

In general, you should implement a raytracer. The raytracer can be run on the command line with a single argument, that is an input file. All parameters are contained in the input file, whose

[1] Hints on Raytracer Design and Classes https://inst.eecs.berkeley.edu/~cs184/fa09/resources/raytracing.htm

[2] Raytracing Implementation Journal https://inst.eecs.berkeley.edu/~cs184/fa09/raytrace_journal.php (One minor note on this resource: our convention is that even if an image is non-square, objects should appear in the correct aspect ratio; a sphere at the center of an image will look like a circle, and not be stretched.)

[3] https://freeimage.sourceforge.io/documentation.html

format is specified on Section 2.2.1. Your raytracer will parse the input file, reading in geometry, materials, lights, transforms etc. It will then raytrace the scene displaying an image. There is no need for user interaction.

Ray tracers are slow

Finally, ray tracers (especially unaccelerated ones like what you are building) tend to be very slow. You may want to display some kind of progress indicator to let one see how much of the scene is done (text printed out is fine). Also, for debugging, always start with low resolution images (say 160 x 120) and make sure things look reasonable before rendering final high resolution (640 x 480 or higher) versions.

The default build option in Visual Studio is Debug, and while this allows for easier debugging, it causes code to run much slower. If you want a speedup after your raytracer is debugged, change the build mode to Release. This will require you to reset your project properties to similar values as in the Debug build mode, as well as copy your FreeImage DLL to the newly created Release directory if you used this image library.

2.1.4 Saving Images

For the purpose of (only) actually writing the output image file, you can reuse any image processing libraries either online or that you have for earlier assignments. It is simplest to use the FreeImage library for writing the output file (the same library that was bundled with homeworks 0,1, 2 and 3). If you feel this would be too much bother, you can also write out ppm files, and convert them offline. Note that this assignment just requires the ability to write an output file, and does not require OpenGL.

FreeImage Library

The simplest way to output your image files is probably to use the FreeImage library which we use to save screenshots. First, compile your ray tracer with the FreeImage library by including FreeImage.h in your solution as well as -lfreeimage in your linker flags. Take a look at the skeletons for each platform to see how to implement this. On OSX/Linux, the library file must be in one of the directories specified in the makefile or in /usr/lib. Also make sure FreeImage.h is in your include directory for either platform. For Visual Studio, you will likely need to bundle the FreeImage DLL with your solution as well as the library/include files, and specify these in the project properties.

Note: FreeImage is already included in your HW3, so you can just reuse that code.

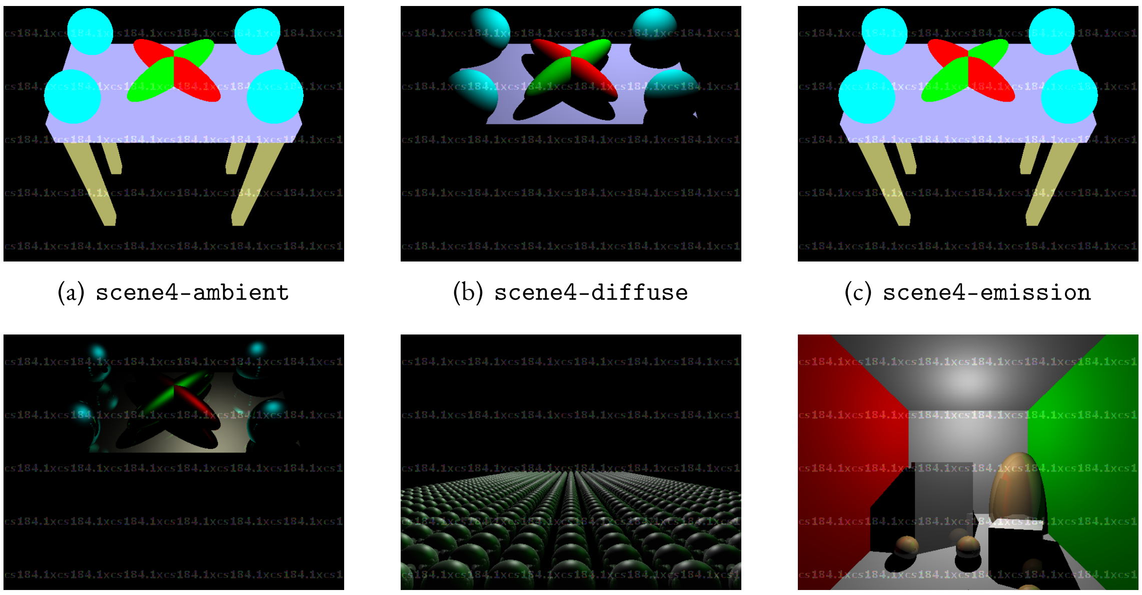
Before you use FreeImage, call `FreeImage_Initialise()`. To record the colors for each pixel, use an array of one byte elements. Then convert this array to a FIBITMAP object as follows, assuming the bytes are in RGB order:

```
FIBITMAP *img = FreeImage_ConvertFromRawBits(pixels, w, h, w * 3, 24, 0xFF0000, 0
    x00FF00, 0x0000FF, false);
```

where `pixels` is a pointer to the beginning of your pixel array, `w` and `h` are the width and height of the image, `w*3` is the number of bytes on one row of pixels, 24 is the number of bits per pixel, the hex values are bit masks for each color R, G and B, and `false` tells FreeImage that the bottom-left pixel is first. If you start with the top-left pixel, change this to true. This method assumes that your pixel array goes one row at a time.

To save your image as a PNG, use the following command:

```
FreeImage_Save(FIF_PNG, img, fname.c_str(), 0);
```



(a) `scene4-ambient` (b) `scene4-diffuse` (c) `scene4-emission`

(d) `scene4-specular` (e) `scene5` (f) `scene6`

Figure 1 Reference solutions.

where `FIF_PNG` specifies the file type, `img` is the pointer you obtained by the function call above, `fname` is a C++ STL string object converted to a C string for the filename (any C string argument works here), and 0 indicates no special flags are used.

Finally, call

```
FreeImage_DeInitialise()
```

to exit cleanly.

There is some example code in the provided parser skeleton code `file.cpp`.

2.2 Specification

2.2.1 Inputfile Format

The input file consists of a sequence of lines, each of which has a command. For examples and clarifications, see the example input files. The lines have the following form. Note that in practice, you would not implement all these commands at once but implement the smallest subset to debug the first aspect of your raytracer (camera control), then implement more commands to go to the next step and so on. This section contains the complete file specification for reference.

- **# comments**: This is a line of comments. Ignore any line starting with a #.
- **Blank line**: The input file can have blank lines that should be ignored.
- **command parameter1 parameter2 ...**: The first part of the line is always the command. Based on what the command is, it has certain parameters which should be parsed appropriately.

We now discuss each of the various commands you need to implement, along with the default values to use where appropriate.

General

Your program is required to read the following general commands

- **size *width height***: The size command must be the first command of the file, which controls the image size.
- **maxdepth *depth***: The maximum depth (number of bounces) for a ray (default should be 5).
- **output *filename***: The output file to which the image should be written. You can either require this to be specified in the input, or you can use a suitable default like stdout or raytrace.png.

Camera

The camera is specified as follows. In general, there should be only one camera specification in the input file; what happens if there are more than one specification can be left undefined. (You can require the camera command to come before any geometry in the file, although it doesn't seem you really need to require that).

- **camera *lookfromx lookfromy lookfromz lookatx lookaty lookatz upx upy upz fov***: specifies the camera in the standard way, as in homework 2. Note that fov stands for the field of view in the y direction. The field of view in the x direction will be determined by the image size. The world aspect ratio (distinct from the width and height that determine image aspect ratio) is always 1; a sphere at the center of a screen will look like a circle, not an ellipse, independent of the image aspect ratio.

Geometry

For this assignment, you will worry only about spheres and triangles. flat triangles, and triangles with prescribed normals. These can be specified in a number of different ways, and differ substantially from what you were asked to do for HW2. There are two types of triangles: flat triangles and triangles with prescribed normals. A flat triangle is the standard triangle described by three vertex positions, and the normals of the triangle is just the constant normal orthogonal to the plane containing the flat triangle. A triangle with prescribed normal is a triangle whose corners are hooked to vertices with a customized normal, and the normals within the triangle are interpolated.

- **sphere *x y z radius***: Defines a sphere with a given position and radius.
- **maxverts *number***: Defines a maximum number of vertices for later triangle specifications. It must be set before vertices are defined. (Your program may not need this; it is simply a convenience to allocate arrays accordingly. You can ignore this command [but still parse it] if you don't need it).
- **vertex *x y z***: Defines a vertex at the given location. The vertex is put into a pile, starting to be numbered at 0.
- **tri *v1 v2 v3***: Create a triangle out of the vertices involved (which have previously been specified with the vertex command). The vertices are assumed to be specified in counter-clockwise order. Your code should internally compute a face normal for this triangle.

Transformations

You should be able to apply a transformation to each of the elements of geometry (and also light sources). These correspond to right-multiplying the modelview matrix in OpenGL and have exactly the same semantics, just like in HW2. It is up to you how exactly to implement them. At the very least, you need to keep track of the current matrix. (Presumably, you can reuse some of the same implementation you did for HW2). For triangles, you might simply transform them to the eye coordinates and store them there. For spheres, you could store the transformation with them, doing the trick of pre-transforming the ray, intersecting with a sphere, and then post-transforming the intersection point. The required transformations to implement are:

- **translate** ***x y z***: A translation 3-vector.
- **rotate** ***x y z angle***: Rotate by angle (in degrees) about the given axis as in OpenGL.
- **scale** ***x y z***: Scale by the corresponding amount in each axis (a non-uniform scaling).
- **pushTransform**: Push the current modeling transform on the stack as in OpenGL. You might want to do pushTransform immediately after setting the camera to preserve the identity transformation.
- **popTransform**: Pop the current transform from the stack as in OpenGL. The sequence of popTransform and pushTransform can be used if desired before every primitive to reset the transformation (assuming the initial camera transformation is on the stack as discussed above).

Note that all of these commands are exactly the same as in HW2.

Lights

Implement the following lighting commands:

- **directional** ***x y z r g b***: The direction to the light source, and the color, as in OpenGL.
- **point** ***x y z r g b***: The location of a point source and the color, as in OpenGL.
- **attenuation** ***const linear quadratic***: Sets the constant, linear and quadratic attenuations (default 1,0,0) as in OpenGL. By default there is no attenuation (the constant term is 1, linear and quadratic are 0; that's what we mean by 1,0,0).
- **ambient** ***r g b***: The global ambient color to be added for each object (default is .2,.2,.2).

Note also that if no ambient color or attenuation is specified, you should use the defaults (you may have used black as a default in HW2; here the defaults are specified). Note that we allow the ambient value to be changed between objects (so different objects will be rendered with a different ambient term; this is used frequently in the examples). Finally, note that here and in the materials below, we do not include the alpha term in the color specification.

Materials

Implement the following material properties.

- **diffuse** ***r g b***: specifies the diffuse color of the surface.
- **specular** ***r g b***: specifies the specular color of the surface.
- **shininess** ***s***: specifies the shininess of the surface.
- **emission** ***r g b***: gives the emissive color of the surface.

2.3 Basic Implementation Steps

What follows below is a step-by-step approach to implementing the requirements for your raytracer. We strongly recommend you proceed in the order below.

2.3.1 Parser

A partially completed skeleton parser is provided in `file.cpp`. You can disregard this file if you prefer your own solution. This parser is nonfunctional, but provides a starting point for your raytracer's file parser. It is recommended that you use it as a base for some sort of scene class which can hold your scene information, or alternatively use global variables (see `extern` in C++). As it stands, the skeleton parser does not return or store the information it reads in a useful way, so you will have to modify it. `file.cpp` also has an example usage of FreeImage, but make sure to call `FreeImage_Initialise()` and `FreeImage_DeInitialise()`.

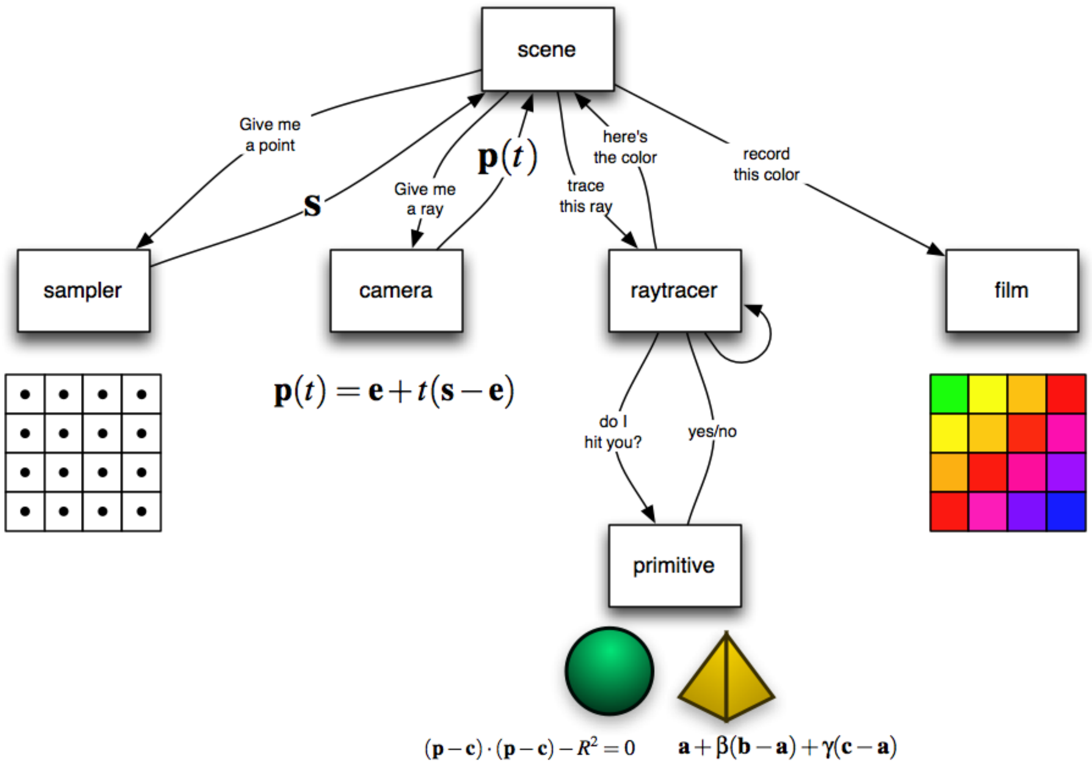


Figure 2 Components of a raytracer program.

2.3.2 Camera

The first step is to implement the camera model. The user should be able to specify the camera, and you should test using a simple scene (initially, it may be simplest if you test using a single quad as geometry, coding up a simple ray-quad intersection test and don't worry about shading). For this part of the assignment, you will need to know how to set a camera, and how to generate corresponding rays for each pixel. Get this part completely debugged before proceeding further.

Please note that the raytracer sends rays through the center of a pixel, i.e. at values 0.5,1.5,... rather than at integers (pixel corners).

2.3.3 Ray-Surface Intersection Tests

Now, you should implement the core of your raytracer, which are the ray-surface intersection tests, in this case for triangles and spheres. You should debug separately with each primitive, making sure things work as expected. You could try images of the second test scene of a dice (from each of the camera positions specified).

If you are testing only the ray-sphere intersection test, it is much simpler to ignore shading and use only an ambient (constant) term to see the intersections in an image.

Next, you should implement transformations, allowing the user to specify transformed geometry. You might want to try the third test scene (the table with ellipses and spheres). Again, shading is not yet important, but you should be sure the core ray-surface intersection tests for geometry are debugged.

2.3.4 Lighting and Shadows

Next, you should implement shading. For this, simply implement the similar shading model as in HW2. In particular, the color at each point is given by

$$I = A + E + \sum_i V_i \frac{L_i}{c_0 + c_1 r + c_2 r^2} \left(D \max(N \cdot L, 0) + S \max(N \cdot H, 0)^s\right) \tag{1}$$

where I is the final intensity, A is the ambient term, E is the self-emission, and the diffuse D and specular S are summed over all lights i with intensity L_i. N is the surface normal, L is the direction to the light, H is the half-angle, and s is the shininess. For ray tracers, there is an additional term V_i which is the (binary) visibility to the light, corresponding to shadows. You should cast a shadow ray to all lights at the intersection point to determine if they are visible (determine V_i). If visible, we simply compute the diffuse contribution as well as the specular contribution. We also include an attenuation model, corresponding to traditional OpenGL (the attenuation model applies only to point lights, not directional lights). c_0, c_1, c_2 are the constant, linear and quadratic attenuation terms, while r is the distance to the light. Physical point lights have $(c_0, c_1, c_2) = (0, 0, 1)$ while the default (no attenuation) is $(c_0, c_1, c_2) = (1, 0, 0)$.

2.3.5 Recursive Ray Tracing

Next, you should implement a recursive ray tracer for mirror reflections. The simplest way of doing it is to shoot a single ray in the mirror direction, weighting its contribution by the specularity or S.

$$\begin{aligned} I = A + E + \sum_i V_i \frac{L_i}{c_0 + c_1 r + c_2 r^2} \left(D \max(N \cdot L, 0) + S \max(N \cdot H, 0)^s\right) \\ + S I_R \end{aligned} \tag{2}$$

where I_R is the color of the mirror reflection ray. Here, the multiplication SI_R of two RGB-values (S and I_R) is the componentwise multiplication. Since this reflected ray may spawn additional reflections, the tracing is recursive, with the maximum depth of the ray tree controlled by the maxdepth parameter.

2.4 Expectation

It is expected that your raytracer will be able to read the scene files provided and produce images that reproduce (or at least are very close to) the reference solutions shown above.) Beyond this demonstration, you can build your own scene to showcase your raytracer.

A possible extra credit is possible if you extend your raytracer to support global illumination (with diffuse color done with recursive random sampling). See the guide document for bonus credit for more detail.

A Debugging and Implementation Notes

These non-official implementation notes are contributed by Fall 2012 student KOlegA. They are likely to be helpful (you also don't need to strictly follow them).

A.1 Own classes for math or glm (or other lib)

Of course with library functions you can implement the assignment faster and maybe they work faster, but writing your own classes (Vector, Normal, Point, Matrix, Color, ...) gives more flexibility and makes the code easier for understanding. Besides, it's not hard to implement them, and they can be faster because they don't have to be so general. For example in multiplying Matrix4x4 by Vector3 you need to consider only matrix3x3. The only complicated thing to implement is Matrix4x4 inversion, but we don't need that for this assignment. By multiplying inverted transform matrices in reverse order, we can obtain the inverted matrix. (That is, simply invert or undo each transformation, but in reverse order: the last transform applied is the first transform inverted or undone). $(ABC)^{-1} = C^{-1}B^{-1}A^{-1}$. Of course, you still need to invert the basic transforms like A^{-1}, but inverting simple translation or rotation is straightforward.

A.2 scene4-ambient and scene4-emission

If transforms and intersections are implemented correctly, the image should look like the image from the reference. At this step you can skip implementing the shader and shadows, and just set color equal to ambient + emission.

Image is smaller than it should be: Probably you are calculating fovx incorrectly. Should be

$$\tan\left(\frac{\text{fovx}}{2}\right) = \tan\left(\frac{\text{fovy}}{2}\right)\frac{\text{width}}{\text{height}} \tag{3}$$

Or the camera ray is not through the center of the pixel (it may be through integers 0, 1, 2 instead of half-integers 0.5, 1.5, 2.5).

Objects overlap incorrectly: Probably you forgot that you need to find the closest intersection object.

A.3 scene4-diffuse

Start from marking shadow areas with solid color.

Image has black points: You forgot to make epsilon shift before shooting the ray from the intersection point to the light source.

Image has color stripes: You forgot to normalize direction somewhere.

Shadows are incorrect: Probably you are transforming normals incorrectly. Review the lecture about transforming normals. If you are applying a matrix M to transform a vector, to transform the normal, you should apply the inverted transposed matrix $M^{-\top}$.

A.4 scene4-specular

Shading is almost like in previous homeworks, except attenuation for point light source and eyepos which was previously zero. Initially variable eyepos equals to camera position, after reflection it should be point of reflection. So, eyepos is ray start point.

A.5 scene5

If all previous scenes work fine and directional lights are implemented correctly, there shouldn't be any problem.

A.6 scene6

Picture black with white cube: Light source could be between two objects. You generated ray from intersection point to the light source and found the intersection, but that intersection was behind the light source, so you should consider only objects between intersection point and light source.