CS 283 Simple Raytracer for students new to Rendering

Ravi Ramamoorthi

This assignment should be done only by those small number of students who have not yet written a raytracer. For those students only, it will be a prelude to the Monte Carlo path tracer in assignment 2, and counts for 30% of the grade of assignment 2. Most students will not do this assignment, but will rather start with existing raytracing code. However, you may still want to look at this as a reference. As in assignment 2, you can (and are encouraged) to work in groups of two, but not required to do so.

This assignment asks you to write a first simple raytracer. Raytracers can produce some of the most impressive renderings, with high quality shadows and reflections. They are also conceptually very simple. However, the actual implementation effort can be considerable. Therefore, you should start early (aim to do so today, and finish in a couple of weeks, so you're ready for the path tracing component in regular assignment 2) and proceed through the assignment strictly in the order of the specifications provided. The assignment can be fairly hard to debug. You should strive to make progress incrementally. Start with the simplest functionality (can you render an image with one triangle on the screen?), debug that fully and then proceed. Trying to write the whole thing at once will lead to a mess of undebuggable code.

Support: Raytracing is well understood and covered in the lecture material, as well as a number of other texts and online materials. Ask the instructor if you want more references on ray-surface intersections and so forth. At this time, we do not provide any skeleton code for this assignment. Again, 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 POVray. As in the previous assignment, I have no objection to your looking at the source code in them for inspiration and understanding, but all code you write must be your own.

The one piece of support provided is an OpenGL previewer. This takes as input the file you will be using as input to your raytracer, and uses OpenGL to create an image. The source code is also available, primarily as an example of the simple parsing I'm doing. It's probably not too helpful for actually ray-tracing. You can use the viewer to prototype scenes that you model quickly and get a coarse comparison with respect to the images in your raytracer. Remember that your raytraced renderings should look a lot nicer with per-pixel shading, shadows and interreflections.

Besides this, we provide 3 example input scenes, along with the output from the OpenGL previewer for the early parts of the assignments. Your submission should include these results. In addition, you will need to model some simple additional scenes to show off the effects from your raytracer.

Finally, for the purpose of actually writing the output image file, you can use any suitable image libraries. If you feel this would be too much bother, you can also write out ppm files, and potentially convert them offline. The format for ppm is very simple; just look up portable pixel map on the web.

1 Turning in the Assignment

This applies only to students who have not previously written a raytracer, and are doing this assignment for 30% credit on assignment 2. In these cases only, you will include a description of this assignment either as part of, or separate from, the regular assignment 2. This should be done in the standard way by e-mailing the instructor a PDF or creating a website that you don't modify after the due date. Your website should also include downloadable source and executable code. You should include a complete description or writeup showcasing your results. Failure to fully document the results of your program will make the graders very unhappy. Being honest, noting what works and what doesn't work for all the features will make the assignment easy to grade and make us very happy.

2 What you need to implement

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 format is specified below. Your raytracer will parse the input file, reading in geometry, materials, lights, transforms etc. It will then raytrace the scene displaying an image. You may augment the file format below with any commands you need for specific effects in your raytracer, provided you document them in your writeup. You also have a little flexibility in modifying this format, if you feel you need to. In that latter case, please check with me beforehand.

Finally, ray tracers (especially unaccelerated ones like what you are building) tend to be very slow. You should 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 160×120) and make sure things look reasonable before rendering final high resolution (640×480) versions.

2.1 File 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 as well as the source code for the OpenGL viewer. 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 subsection 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.

2.1.1 General

You should implement the following general commands:

1. *size width height*: The size command must be the first command of the file, which controls the image size.
2. *maxdepth depth*: The maximum depth (number of bounces) for a ray (default should be 5).
3. *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.bmp

2.1.2 Camera

The camera is specified as follows. In general, there should be only one camera specification in the input file; what happens if there is more than one specification can be left undefined[1].

1. *camera lookfromx lookfromy lookfromz lookatx lookaty lookatz upx upy upz fov*: The first set of parameters (except fov) is exactly what is specified to gluLookAt, which you should be familiar with. The next parameter controls the field of view, as in the parameter to gluPerspective. If you are confused about this, read the (very straightforward) implementation in the OpenGL viewer.

[1] Because of the structure of the OpenGL viewer, it requires the camera to be specified before any geometry; it is fine if you require a similar condition, though this is likely to be irrelevant for raytracing.

2.1.3 Geometry

For this assignment, you will worry only about spheres and triangles. These can be specified in a number of different ways. The first set of commands you need to implement are as follows:

1. *sphere x y z radius*: Defines a sphere with a given position and radius.
2. *maxverts number*: Defines a maximum number of vertices for later triangle specifications. It must be set before vertices are defined.
3. *maxvertnorms number*: Defines a maximum number of vertices with normals for later specifications. It must be set before vertices with normals are defined.
4. *vertex x y z*: Defines a vertex at the given location. The vertex is put into a pile, starting to be numbered at 0. This is very similar to the OFF file format in assignment 2.
5. *vertexnormal x y z nx ny nz*: Similar to the above, but define a surface normal with each vertex. The vertex and vertexnormal set of vertices are completely independent (are as maxverts and maxvertnorms).
6. *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.
7. *trinormal v1 v2 v3*: Same as above but for vertices specified with normals. In this case, each vertex has an associated normal, and when doing shading, you should interpolate the normals for intermediate points on the triangle.

2.1.4 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. It is up to you how exactly to implement them. At the very least, you need to keep track of the current matrix. 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:

1. *translate x y z*: Translate to a given position.
2. *rotate x y z angle*: Rotate by angle (in degrees) about the given axis as in OpenGL.
3. *scale x y z*: Scale by the corresponding amount in each axis.
4. *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.
5. *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).

2.1.5 Lights

You should implement the following lighting commands. With respect to comparison to the OpenGL model, lights here have a single color for both diffuse and specular components.

1. *directional x y z r g b*: The direction to the light source, and the color, as in OpenGL.
2. *point x y z r g b*: The location of a point source and the color, as in OpenGL.

3. *attenuation const linear quadratic*: Sets the constant, linear and quadratic attenuations (default 1,0,0) as in OpenGL.
4. *ambient r g b*: The global ambient color to be added for each object (default is .2,.2,.2)

2.1.6 Materials

Finally, you need to implement the following material properties.

1. *diffuse r g b*: Diffuse color as in OpenGL.
2. *specular r g b*: Specular color as in OpenGL.
3. *shininess s*: Shininess as in OpenGL.
4. *emission r g b*: Emissive color as in OpenGL.

2.2 Writeup

Some points will be allocated for a simple and clear writeup that documents your project. For now, keep the scenes you model relatively simple. The focus should be on showing the technical achievements of your raytracer, rather than on the modeling.

2.3 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. You should in particular, include the images corresponding to the first test scene which shows a single quad (for now, it's acceptable if you code 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.

2.4 Ray-Surface Intersection tests

Now, you should implement the core of your raytracer, which are the ray-surface intersection tests, in this case case for triangles and spheres. You should debug separately with each primitive, making sure things work as expected. In your writeup, at least include the images of the second test scene of a dice (from each of the camera positions specified)[2].

Next, you should implement transformations, allowing the user to specify transformed geometry. You should at the least show the results of running on 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.

2.5 Lighting and Shadows

Next, you should implement shading. For this, simply implement the OpenGL shading model essentially. In particular, the color at each point is given by

$$C = K_a + K_e + \sum_{i=1}^{n} S_i L_i \left(K_d \max(l_i \cdot n, 0) + K_s (n \cdot h_i)^{shininess} \right), \quad (1)$$

where K_a is the global ambient value, K_e is the emission for the surface, and the rest involves diffuse and specular sums over all lights. S_i is a binary shadowing term for light i. You should cast a shadow ray to all lights at the intersection point to determine if they are visible. If visible, we simply compute the diffuse

[2] Shading is not too important for now. I've just used a separate ambient color for each face. It's not clear it makes sense for your raytracer to follow this trick; you might imagine adding a command to just explicitly set the color for each face without worrying about lighting for now.

contribution (K_d is the surface diffuse color, L_i is the light color, and l_i is the light direction, n is the surface normal) as well as the specular contribution (K_s is the surface specular color, while h_i is the half-angle for that light). Note that, unlike standard OpenGL, you will compute the shading at each pixel, and take shadows into account, leading to images that should look much nicer than those produced by the previewer.

To document this part, show some nice scenes which bring out the various shading effects and shadows. The onus is on you to create good scenes. If you are completely lost in terms of modeling, use the example in scene 3, adding lights and materials to bring out interesting effects.

2.6 Recursive Ray Tracing

Finally, you should implement a recursive ray tracer for mirror reflections. You will need to model a nice scene to show this effect off (some limited form of textures like a checkerboard would help here). For inspiration, look at some of the examples in class; glass spheres are always a nice object for ray tracing. For this part, you can modify the file format if you think it's appropriate [for instance adding a mirror material; perhaps if shininess is set to 0]. The simplest way of doing it is to shoot a single ray in the mirror direction, weighting its contribution by the specularity or K_s. 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.