COMS W4160— Computer Graphics Spring 2017

Programming Assignment 3

out: Mar. 20, Monday
due: Apr. 2nd, Sunday 10:00PM. No extension!

Ray tracing is a simple yet powerful algorithm for rendering photorealistic images. Within the accuracy of the scene and shading models and with enough computing time, the images produced by a ray tracer can be physically accurate and appear indistinguishable from real images[1]. In this assignment, you will implement several simple rendering algorithms. Successful implementation will be able to render basic geometry and complex mesh models (given as triangle meshes) in a photorealistic way.

1 Getting Started

The devil is in the details! To help you quickly start due to this semester's tight schedule, I prepared a starter code to save you from spending time on coding up support functionalities. Please build your code on the starter code. But before you start coding, **make sure that you look through the starter code and have a complete picture of the core rendering techniques**. Please try to map what we discussed in class to individual components of the starter code. This will help you understand the rendering algorithm much deeper and ease your programming.

Java Starter Code. We have prepared a Java starter code including all components but the Monte Carlo integration. The starter code also comes with a simple ambient occlusion renderer to allow you play with the code.

1.1 Trying the Starter Code

With the Java starter code downloaded, you can compile it using `javac` or whatever IDEs you prefer. Another suggested option is to use Apache Ant command-line tool. If you have `Ant` installed, simply unzip the file and run the command:

```
unzip aa3.zip && cd aa3 && ant
```

It will compile the code and start to run a test example (see Figure 1). To render a specific scene, use the command like this

```
java -cp carbine.jar ray.RayTracer scene/cbox.xml
```

where `scene/cbox.xml` is an XML file to describe the rendered scene (see Section 1.2). While the rendering is running, a window will appear to show you the rendering progress. After the rendering is finished, a PNG file will be stored in the sample folder of your scene file. In the above example, the generated PNG file will be `scene/cbox.xml.png`.

Realistic image rendering is computationally expensive in general. A high quality image can take hours to render. We provide a parallel renderer that allows you to use all available processors on your computer. To start a parallel rendering, use the command:

[1] For famous historic examples, see http://www.graphics.cornell.edu/online/box/compare.html

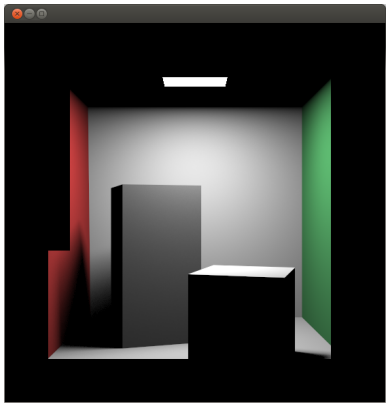


Figure 1: **The ray tracer rendering direct illumination in progress**

```
java -cp carbine.jar ray.ParaRayTracer scene/cbox.xml
```

If you want to manually specify the number of processors for rendering, use the command option '-s', for example, the command

```
java -cp carbine.jar ray.ParaRayTracer -s 4 scene/cbox.xml
```

uses four processors for parallel rendering.

The default rendering algorithm I implemented as an example for you is ambient occlusion (see Figure 2a), which is a fast global method widely used in architecture and geometry design. The file ray/renderer/AmbientOcclusion.java illustrates the basic usage of the data structures and functions for rendering.

1.2 Scene Configuration File

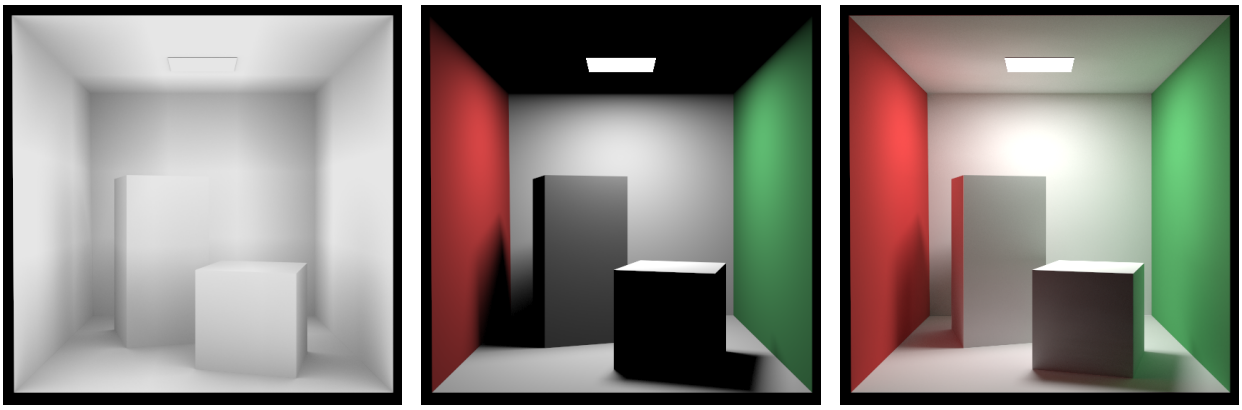
A rendered scene is described by an XML file (see scene/cbox.xml as an example). After you implement your own rendering algorithm, you will change the renderer element in the file to point to your rendering class. For example, if you specify your renderer as

```
<renderer type="DirectOnlyRenderer">
</renderer>
```

The program will use the render class ray.renderer.DirectOnlyRenderer.class with the light sampling method specified in ray.renderer.ProjSolidAngleIlluminator.class. This XML file will be loaded and parsed by ray.io.parser.class. In Appendix A, I will present more details about it.

2 Programming Requirements

To help you clearly locate where to code in this large project, I have marked all the functions or parts of classes you need to complete with TODO in the source code. If the code is marked as TODO (A), that code should be implemented first to help you understand the code. If it is marked as TODO (B), it requires the understanding of global illumination and needs more involved coding. The following explanations are to give you an overview of the tasks. More instructions are provided in the comments of the source code.



(a) Ambient occlusion rendering (b) Direct illumination rendering (c) Global illumination rendering

Figure 2: **Different rendering algorithms** can be implemented by inheriting the ray.renderer.Renderer interface, and put the implementation file in ray/renderer.

1. ray/surface/Sphere.java (TODO (A)): This class is a subclass of Surface, representing a sphere geometry. It contains center, a 3D point, and radius, a real number. The only function you need to complete is the intersect method that checks if a given light ray intersects with the sphere. Intersection results will be stored in the IntersectionRecord variable.

2. ray/math/Geometry.java (TODO(A)): Next, you implement the algorithm that uniformly sample the projected solid angles. This is the sampling algorithm that we talked about in class.

3. ray/renderer/DirectOnlyRenderer.java (TODO(A)): This class implements the simple stochastic rendering algorithm with only direct illumination considered. That is, the surface can be illuminated when only the light rays can reach it directly, and light reflections are ignored completely. However, the resulting image is capable of capturing interesting effects such as soft shadows (see Figure 2b as an example). Please test your program using the scene file scene/cbox-direct.xml and submit the resulting image.

4. ray/renderer/BruteForcePathTracer.java (TODO(B)): This class implements the rendering algorithm that considered ray reflections on the object surface. Theoretically, a ray can bounce on object surfaces many times, resulting in a impractical simulation process. You can terminate the recursive ray tracing according to the parameter depthLimit in ray.renderer.PathTracer.java. This of course is not physically accurate, but it is sufficient to produce realistic images like the one in Figure 2c. Please test your program using the scene file scene/cbox-global.xml and submit the resulting image.

   Note: This part requires understanding of global illumination. Even if your code does not work for this requirement, we will try to look at your code and give partial points if some part is implemented correctly. Therefore, make sure you carefully comment the code so our TAs can understand it. After all, it is your responsibility to deliver us readable code.

2.1 Extension for More Bonus

The starter code can be extended to provide more advanced features very easily. You can optionally implement some extensions chosen from the following list.

1. **Texture mapping**: Write a new material class that implements the Material interface and provide the functionality of image texture mapping.

2. **Cube-mapped background**: A ray tracer need not return black or a constant ambient color value when rays do not hit any objects. Commonly, background images are supplied that cover a large cube surrounding the scene. The ray that does not intersect objects are used to look up color values in these images. This technique is called *cube-mapping*. You need to implement a new background class in ray/background/Cubemap.java. Part of the code is already there for your reference.

3. **Propose your own**: You can propose your own extension based on something you heard in lecture, read in the book, or learned about somewhere else. Doing this requires a little extra work to document the extension and come up with a test case to demonstrate it.

3 Submission and FAQ

Submission Checklist: Submit your assignment as a zip file via courseworks. Your submission must consist of the following parts:

1. **Rendered images:** Please include the resulting images of the scene files in the starter code. In addition, use your imagination and create **two** other scenes and render them using your DirectOnlyRenderer. If your BruteForcePathTracer works, please also include two rendered images using your BruteForcePathTracer. Please submit all the images and your created scene files. The image size should be no smaller than 300 pixels×300 pixels.

2. **Documented code:** Your code should be reasonably documented and be readable/understandable by the TAs. If the TAs are not able to run and use your program, they will be unable to grade it. To ensure the TAs can grade your assignment, please make sure your code can be compiled successfully. Please also submit your scene files (.XML) that are used to generated the images in your submission.

3. **Brief report:** Include a description of what youve attempt, special features you have implemented, and any instructions on how to run/use your program. In compliance with Columbia's Code of Academic Integrity, please include references to any external sources or discussions that were used to achieve your results.

4. **Additional results (optional):** Please include additional information, pictures, videos, that you believe help the TAs understand what you have achieved.

Note: In order to generate high-quality rendered images, you might need to increase the number of samples per pixel (specified in the scene file). This will lead to smaller variance of your Monte Carlo integration, and thus less noise. But that will take a relatively long time. When you debug, it may be a good idea to reduce the image resolution and/or the number of samples per pixel. So your rendering terminates much faster, providing you a shorter tuning cycle.

Evaluation. Please make sure your code can be compiled successfully on your machine. Your work will be evaluated on how well you demonstrate proficiency with the requested functionalities, the correctness of the resulting images, and the quality of the submitted code and documentation.

A ray.io.Parser

The Parser class contains a simple and, we like to think, elegant parser based on Java's built-in XML parsing. The parser simply reads a XML document and instantiates an object for each XML entity, adding it to its containing element by calling set... or add... methods on the containing object.

For instance, the scene file

```
<scene>
  <image>560 560</image>

  <sampler type="JitteredSampler">
    <numSamplesU>2</numSamplesU>
    <numSamplesV>2</numSamplesV>
  </sampler>

  <renderer type="AmbientOcclusion">
    <length>0.2</length>
  </renderer>

  <light type="PointLight"> <location>0. 510. 0</location> </light>

  <camera>
    <eye>278.0 273.0 -800.0</eye>
    <target>278.0 273.0 0</target>
    <up>0 1 0</up>
    <yFOV>40</yFOV>
  </camera>

  <material name="red" type="Homogeneous">
    <brdf type="Lambertian">
      <reflectance>0.6 0.05 0.05</reflectance>
    </brdf>
  </material>
  <surface type="Sphere">
    <material ref="red" />
    <center>300. 200. -400.</center>
    <radius>100.</radius>
  </surface>

  <background type="Uniform"> <radiance>0.0 0.0 0.0</radiance> </background>
</scene>
```

results in the following construction sequence:

1. Create the scene.
2. Set the output image resolution as 560×560.
3. Set to use the sampling algorithm ray.sampler.JitteredSampler.

4. Set to use the rendering algorithm ray.renderer.AmbientOcclusion.
5. Add a point light source at the position $(0, 510, 0)$.
6. Set up the camera parameters just like what you did in OpenGL.
7. Add one Lambertian material named as "red".
8. Add a geometry Sphere using the previously specified material "red".
9. Use the uniform background color (black) which is the color when a ray does hit anything.

Which elements are allowed where in the file is determined by which classes contain appropriate methods, and the types of those methods' parameters determine how the tag's contents are parsed (as a number, a vector, etc.). There is more detail for the curious in the code.

B Triangle Mesh File Format

The current implementation use a simple triangle mesh file format. The format is a text file, staring with the number of vertices and the number of triangles followed by the vertex positions, triangle indices and optionally texture coordinates and normals. You can design your geometry using any 3D software, export it into any text file format (e.g., wavefront obj) and write a small script to transform it into the .msh file format used in the code. A bunny mesh file is also provided in the scene directory. For your convenience, I also provide a simple python script obj2msh.py to generate the .msh file from a provided wavefront .obj file.