CS52 Assignment 1: Raytracing I

Out: Oct 1. Due: Oct 15.

Introduction

In your first assignment, you will create a simple ray tracer. Ray tracing is a very powerful algorithm capable of creating complex and realistic images. While you will not be able to generate realistic images in your first assignment, your code will create a lot of interesting effects.

You are to perform this assignment using Java. To ease your development, we are providing a set of classes to load a scene, perform basic mathematical calculations and save your image results.

Also you can use the layout of the code as indication of how to structure your raytracer. We have removed the code of each of the functions you are expected to fill in. If you feel this structure does not suit your needs, feel free to change it, but please leave comments as to what your new functions are meant to do to help us evaluating your code in case of errors.

The final result of this assignment is a simple ray tracer with support for the following features:

- Algorithm
  - Basic ray tracer
  - Superampled ray tracer to reduce jaggies
  - Shadows
  - Reflections
- Geometry
  - Spheres
  - Triangles
- Materials
  - Lambertian
  - Phong
  - ShinyPhong (i.e. Phong with a mirror reflection)
- Lights
  - Point lights

Requirements

1. Implement a basic raytrace algorithm that will sample the image plane and generate a picture. Your algorithm should support shadows and reflections. You are not required to implement refraction.

2. Supply the camera class with a method to generate rays, `Camera.generateRay`, given an image plane position specified in normalized coordinates, i.e. in $[0, 1] \times [0, 1]$. You can do this by first computing the image plane point (by filling in `Camera.imagePlanePoint`) and then generating the direction given this point and the camera position, as shown in class.

3. Ray-Scene intersection code. Add code to the `Scene.intersect` to support intersection of multiple primitives.

4. Ray-Sphere/Triangle intersection code. Add code to the `Sphere.intersect` and `Triangle.intersect` to support intersection of spheres and triangles.

5. Add code to the `Lambert`, `Phong` and `ShinyPhong` class to support shading. In particular the method ... should be an implementation of the shading equation given in class for each material. `Lambert` and `Phong` do not contain reflection, while for `ShinyPhong`, implement the reflection routines described below (note that it derives from `Phong`, so there is no need to implement the other routines).

6. Implement point lights with no falloff, i.e. their intensity will be the same regardless of the surface position.

7. Implement an antialias raytracer by sampling multiple times the image for each pixel. Follow the pseudocode given in class for this.

8. Do not worry about malformed input. For example your camera will never be placed inside a sphere. Also do not worry about hitting the backside of surfaces.

Framework Overview

We suggest you use our framework to create your raytracer. We have removed from the code all the function implementations you will need to provide, but we have left the function declarations which can aid you in planning your solution.

Following is a brief description of the classes provided in the raytracer. We have taken various steps to simplify the framework code as much as possible. In particular, you will notice that most of the variables are declared as public and all classes are in the same package. Please note that this is not good programming practice! We have done this to help making the code more readable by reducing the length of the classes.

Main. This is the main entry point of your code. It will loop over all the scene files given, invoke the raytracer to generate two images, a fast one and one with antialiasing. It will then save this images to disk as PNG files. You can run the code using `Main sceneFile1 sceneFile2 ....`

RayTracer. This is the class that implements the raytracing algorithm. It has two entry points you should fill in: `render` that will compute the image using one sample per pixel and `rendersuperSampled` that will compute the image using $nSamples \times nSamples$ samples for each pixel.

The `RayTracer` class also contains the signature of the methods we used in our implementation. You are welcome to use this same structure if you feel it is appropriate. In particular, `computeColor` will test for scene intersection and return `computeIllumination` if the ray hits or black otherwise. `computeIllumination` invokes lights and materials to compute the sample color, calling `computeShadow` to check for shadowed light; it is also responsible for reflections by recursively calling `computeColor` when needed.

FileFormat. This class implements methods to load the scene and save images. The file format is based on a very simple XML format specification documented in the code. The loader should work also on classes you add. Note: to help with debugging, we have removed gamma correction when saving images; this way you can use the image as a way to output values in the [0,1] range. Please ask us if something is not clear with this class.

Scene. The scene is a container for objects, lights and a view camera. It also contains a method `Scene.intersect` that will intersect all the objects and return the closest intersection.

Camera. This is a representation of the scene viewer. It is defined as an origin, a frame of reference, the field of view covered by the image plane and the image size. The frame is oriented as the diagram shown in the class, i.e. the viewer looks along the $-z$ axis, while the x ad y axis are parallel to the image plane.

Surface, Triangle, Sphere. These classes represent the geometry available to construct the scene. Surface is an abstract class defining the interface between the raytracer and the surface classes. Each surface has a material and a method `Surface.intersect` that you should fill in with the intersection code for the particular primitive. Note that the intersection method should check for the ray limits described below.

Material, Lambertian, Phong, ShinyPhong. These classes represent the materials types for the scene. Each material has three methods used to evaluate its light response defined in the `Material` abstract class. In particular `Material.computeDirectLighting` computes the surface response given a surface normal, a lighting direction (pointing away from the surface) and an incoming direction (pointing into the surface - pay attention to this). You should fill in the code for this for the `Lambert` and `Phong` cases. The other interfaces deal with reflections. In particular, `Material.hasReflection` should return true only if a reflection exist for the particular incoming ray and normal. If so, the reflection coefficient is then returned by `Material.computeReflection`.

Light, PointLight. The light classes represent the lights in the environment. `Light` is the abstract class specifying the interface between the lights and the renderer. The interface is comprised of three functions: `Light.computeLightDirection` returns the light direction (pointing toward the light) from where illumination comes, `Light.computeLightIntensity` returns the light color and `Light.computeShadowDistance` returns the maximum distance a ray should travel when checking for shadows. You should implement these functions for `PointLight`, which represents a point light, as define in class (do not add any falloff to this class).

Vec3, Color, ColorImage. These are classes for the basic mathematical types used in the framework. `Vec3` represents three dimensional points, vectors and normals. `Color` represents an (R,G,B) color. `ColorImage` stores a color image.

Ray, Intersection. The `Ray` class represents a ray cast in the scene in its parametric form, i.e. by its origin and direction. It also holds the limits on the ray parameter that the intersection routines should obey to and the depth of a ray measured as the number of times the ray has recursively being bounced around for reflection and refractions.

Hints

- We suggest to implement the raytracer following the same steps presented in class. Start simple, by implementing camera ray generation, raytracer sampling, scene intersection and sphere intersection test. Use one of the given scenes to check this code.
  Now add and test the triangle intersection. Once you are done with it, you can try to test the scene intersection and make sure you are getting this right.
  Now add shading code, by implementing the materials and light classes. Once you are done add shadows and reflection.
  Finally add the supersampling code to remove the jaggies for your images.
- Remember that the framwork code we are supplying is for your own benefit. If you feel you should write code differently, go ahead and do so. The important things is for you to get the images right.
- Check your output with the scenes and results attached to the framework. Add new scene as needed to debug specific bugs. Also, use images to store debugging information. Think of them as the printout statement of your renderer.

Submission

Please send your code and compiled class files to `cs52@cs.dartmouth.edu`. We will run your code by calling the main method provided. Also add the images rendered from the scenes provided.

Extra credit

1. Add a new cylinder primitive, composed of a cylinder lateral surface and two disk caps. Demonstrate your code with a scene containing a Phong cylinder.
2. Add refraction to your code by augmenting the material interface, the raytracer and by adding a new `Glass` material with reflections, refractions and a Phong highlight. You can find the detail on how to generate refraction rays in Shirley's book. The reflection code is a good start for this. Demonstrate your code with a new scene, which should contain a glass sphere.