# CS 4620 Programming Assignment 1
# Ray 1: Ray Tracing Part 1

out: Wednesday 4 September 2013
**due: Wednesday 18 September 2013**

## 1 Introduction

Ray tracing is a simple and powerful algorithm for rendering 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 can appear indistinguishable from real images [1].

The basic idea behind ray tracing is to model the movement of photons as they travel along their path from a light source to a surface to a viewer's eye. Tracing forwards along this path is difficult, since it can be unclear which photons will make it to the eye, especially when photons can bounce between objects several times along their path (and in unpredictable ways). To simplify the system, these paths are computed in reverse; that is, rays originate from the position of the eye. Color information, which depends on the scene's light sources, is computed when these rays intersect an object, and is sent back towards the eye to determine the color of any particular pixel.

In this programming assignment, you will complete an implementation of a ray tracer. It will not be able to produce physically accurate images, but later in the semester you will extend it to produce nice looking images with many interesting effects.

We have provided framework code to save you from spending time on class hierarchy design and input/output code, and rather to have you focus on implementing the core ray tracing components. However, you also have the freedom to redesign the system as long as your ray tracer meets our requirements and produce the same images for given scenes.

## 2 Requirement Overview

A ray tracer computes images in *image order*, meaning that the outer loop visits pixels one at a time. (In our framework the outer loop is found in the class `SimpleRayTracer`.) For each pixel it performs the following operations:

1. *Ray generation*, in which the viewing ray corresponding to the current pixel is found. The calculations depend on the characteristics of the camera: where it is located, which way it is looking, etc. (In our framework, ray generation happens in the subclasses of `Camera`.)

[1] Cornell pioneered research in accurate rendering. See http://www.graphics.cornell.edu/online/box/compare.html for the famous Cornell box, which exists in Rhodes Hall.

2. *Ray intersection*, which determines the visible surface at the current pixel by finding the closest intersection between the ray and a surface in the scene. (In our framework, ray intersection is handled by the `Scene` class.)

3. *Shading*, in which the color of the object, as seen from the camera, is determined. Shading accounts for the effects of illumination and shadows. (In our framework, shading happens in the subclasses of `Shader`.)

In this assignment, your ray tracer will have support for:

- Spheres and axis-aligned boxes
- Lambertian and Phong shading
- Point lights with shadows
- Parallel and perspective cameras

Your ray tracer will read files in an XML file format and output PNG images. We have split ray tracing in 4620 into two parts. For now, you will implement "basic ray tracing." Later in the semester, after you have learned many more graphics techniques, you will implement other features, such as advanced shaders, triangle meshes, and transformations on groups of objects. In this document, we focus on what you need to implement the basic elements of a ray tracer.

# 3 Getting Started

We recommend using the Eclipse IDE for programming assignments in this class (though you are free to use any other editor if you'd prefer). If you haven't already, download Eclipse from http://www.eclipse.org/downloads/. After getting familiar with the environment, open this project by selecting File->Import... and then General/Existing Projects Into Workspace. Navigate to the project on your local machine and select the *ray1-student* folder. If everything went smoothly, you should see the ray1 package in the Package Explorer.

# 4 Specification and Implementation

We have marked all the functions or parts of the functions you need to complete with `TODO` in the source code. To see all these TODO's in Eclipse, select *Search* menu, then *File Search* and type "`TODO`."

## 4.1 Cameras

The camera in the scene is represented by an instance of the `Camera` interface whose specification is in *Camera.java*. Each instance of `Camera` interface must implement the

```
void getRay(Ray outRay, double u, double v);
```

method, which modifies `outRay` to be a ray coming out from the camera. The generated ray is parameterized by two numbers, $u$ and $v$, both ranging from 0 to 1. These numbers represent a coordinate on the image plane, with $(u, v) = (0, 0)$ indicating the lower-left corner of the viewing window, and $(u, v) = (1, 1)$ indicating the upper-right corner.

You will have to implement two classes representing cameras: the `ParallelCamera` and the `PerspectiveCamera`, located in *ParallelCamera.java* and *PerspectiveCamera.java* respectively. Both classes have a number of common fields, located in *Camera.java*:

- `viewPoint`, a 3D point that specifies the position of the eye.
- `viewDir`, a 3D vector that specifies the direction toward which the camera is looking. Its magnitude is not used.
- `viewUp`, a 3D vector that is used to determine the "up" direction as described below.
- `projNormal`, a 3D vector that specifies the normal to the projection plane. By default it is equal to the view direction. Its magnitude is not used.
- `viewWidth` and `viewHeight`, two real numbers that give the dimensions of the viewing window on the image plane.

The `PerspectiveCamera` has one extra field:

- `projDistance`, a real number $d$ giving the distance from the viewpoint to the center of the viewing window.

For both camera classes, you should first set up the camera coordinate system in the `initView` method. The coordinate system consists of three basis vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$. The three vectors should be set up so that:

- $\mathbf{w}$ should point in the exact opposite direction of `projNormal`.
- $\mathbf{u}$ should be the parallel to the image's $u$ (horizontal) axis.
- $\mathbf{v}$ should be parallel to the image's $v$ (vertical) axis.

Refer to sections 2.4.6 and 2.4.7 in the text for more detail on setting up this coordinate system.

After initializing the coordinate system, you should implement the `getRay` method, using this coordinate system to make the calculations simple. For the parallel camera, all generated rays should be parallel to the `viewDir`. The origin of any generated ray should be on a plane spanned by $\mathbf{u}$ and $\mathbf{v}$ passing through `viewPoint`. The view window corresponds to the rectangle such that (1) its width in the $\mathbf{u}$-direction is `viewWidth`, (2) its height in the $\mathbf{v}$-direction is `viewHeight`, and (3) its center point is `viewPoint`.

For example, suppose the camera is at the origin, i.e. `viewPoint` $= (0, 0, 0)$ and is pointing in the direction `viewDir` $= (0, 0, -1)$. Let `viewUp` be $(0, 1, 0)$ and suppose `viewWidth` $=$ `viewHeight` $= 2$. The point $(u, v) = (0, 0)$ should produce a ray with an origin of $(-1, -1, 0)$ and a direction of $(0, 0, -1)$.

For the perspective camera, the view window corresponds to a rectangle on a plane perpendicular to `projNormal` but at distance `projDistance` from `viewPoint` in the direction of `viewDir`. A ray with its origin at `viewPoint` going in the direction of `viewDir` should intersect the center of the image plane. The view window is still of the same size as that of the orthographic camera. Given $u$ and $v$, you should compute a point on the rectangle corresponding to $(u, v)$, and create a ray from `viewPoint` that passes through the computed point.

Refer to sections 4.3.1 and 4.3.2 if you get stuck. Once you have completed the `getRay` methods for both classes, try to test your code as much as possible to catch any mistakes before continuing. If you're confident you've implemented everything correctly, complete the `renderImage` method of the `BasicRayTracer` class so that the camera generates the correct ray for each pixel.

## 4.2 Surfaces

An instance of the `Surface` class represents a piece of geometry in the scene. It has to implement the following method:

- `boolean intersect(IntersectionRecord outRecord, Ray ray)`

  Intersect the given ray with the surface. Return true if the ray intersects the geometry and write relevant information to the `IntersectionRecord` structure. Relevant information includes the position of the hit point, the surface normal at the hit point, the ray parameter at the hit point, and the surface being hit itself.

We ask that you implement the following surfaces:

1. `Box`. This class is a subclass of Surface. It contains `minPt` and `maxPt`, which are 3D points. If the two points are $(x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and $(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$, then the box is given by $[x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}] \times [z_{\text{min}}, z_{\text{max}}]$.
2. `Sphere`. The class contains a 3D point `center` and a double `radius`. See section 4.4.1.

## 4.3 Scene

The `Scene` class stores information relating to the composition, i.e. the camera, a list of surfaces, a list of lights, etc. You need to implement the following methods:

```
boolean getFirstIntersection(IntersectionRecord outRecord,
       Ray rayIn);
boolean getAnyIntersection(Ray ray);
```

The first should loop through all surfaces in the scene, looking for intersections along `rayIn`. This is used when a ray is first cast out into the scene. It should only record the first intersection that happens along `rayIn`, i.e. the intersection with the lowest value of $t$. If there is such an intersection, it is recorded in `outRecord` and true is returned; otherwise, the method returns false, and `outRecord` is not modified.

The second is very much like the first, but a bit simpler and faster. Here, we are only concerned with finding any intersection along the ray rather than the first. Since this is all we are concerned

with when computing shadows, it makes sense to include this as its own method. There is no `outRecord` to modify; simply return true if the ray intersects any surface in the scene, and false otherwise.

### 4.4 Shaders

A shader is represented by an instance of the `Shader` class. You need to implement the following method:

```
void shade(Color outIntensity, Scene scene,
       workspace Workspace),
```

which sets `outIntensity` (the resulting color) to the fraction of the light that comes from the incoming direction $\omega_i$ (given by `incoming`) and scatters out in direction $\omega_o$ (given by `outgoing`).

We ask you to implement two shaders:

- `Lambertian`. This class implements the perfectly diffuse shader. Its diffuse color ($k_d$) is specified by the `diffuseColor` field. The `shade` function should set `outIntensity` to:

  $$k_d \max(\mathbf{n} \cdot \omega_i, 0)$$

  where $\mathbf{n}$ is the normal at the intersection point. See section 4.5.1.

- `Phong`. This class implements the Blinn-Phong lighting model. It has the `diffuseColor` ($k_d$), `specularColor` ($k_s$), and `exponent` ($p$) fields, where the `exponent` is the shininess. The `shade` function should set `outIntensity` to:

  $$k_d \max(\mathbf{n} \cdot \omega_i, 0) + k_s(\max \mathbf{n} \cdot \mathbf{h})^p$$

  only if $\mathbf{n} \cdot \omega_i \geq 0$, and sets `value` to 0 otherwise. Here, $\mathbf{h} = (\omega_i + \omega_o)/\|\omega_i + \omega_o\|$. See section 4.5.2.

Notice that these values should be computed once for each light in the scene, and the contributions from each light should be summed, unless something is blocking the path from the light source to the object. *Shader.java* contains a useful `isShadowed()` method to check this. Overall, the code for each `shade` method should look something like:

reset `outIntensity` to $(0, 0, 0)$
**for** each light in the scene **do**
  **if** `!isShadowed(...)` **then**
    compute color contribution from this light
    add this contribution to `outIntensity`
  **end if**
**end for**

### 4.5 Finishing up

There's one final step before you can start producing images: you need to complete the `shadeRay` method of the `BasicRayTracer` class. This method should compute the outgoing ray and dispatch the color calculation to the appropriate shader. Details are included in the source code; the method itself should only be a few lines.

After you have implemented this feature, use the ray tracer to render the `BasicRayTracer` test scenes. Details on how to do this is given in Appendix A.

## 5 What to Submit

Submit a zip file containing your solution organized the same way as the code on CMS. Include a readme in your zip that contains:

- You and your partner's names and NetIDs.
- Any problems with your solution.
- Anything else you want us to know.

## 6 Appendix A: Testing

The `data` directory contains scene files and reference images for the purpose of testing your code. The `data/scenes/basic_ray_tracer` directory is split into a set of `simple` tests and `complex` tests.

Right click on the `BasicRayTracer` file in Package Explorer, and choose Run As->Java Application. Nothing will happen, since you need to provide the scene as a command line argument. This can be done from the Run dialog (Run->Run Configurations...). Choose the Arguments tab, and provide your command line arguments in "Program arguments".

Note that `BasicRayTracer` prepends "data/scenes/basic_ray_tracer" to its arguments. This means that you only need to provide a folder name and scene file name to run a ray tracer. For example, if you give the argument "simple/four-spheres.xml" to the `BasicRayTracer` it will load the scene file "data/scenes/basic_ray_tracer/simple/four-spheres.xml" and produce an output image in the same directory as the scene file.

You can also specify a list of scene files and the ray tracer will render all of them in sequence. Alternatively you can just give a directory name (e.g. "simple") and it will render all scenes in that directory.

Compare the images you have with the reference images, if there are visually significant differences, consider checking bugs in your code.

# 7 Appendix B: Extra Credit

The parser supports another type of surface, the cylinder. Create a subclass of `Surface`, similar to the `Box` and `Sphere` classes. The class should contain `center`, a 3D point, and `radius`, `height`, both real numbers. Assume that the cylinder is capped, i.e., the top and bottom have caps (it is not a hollow tube). Also, assume it is aligned with the z-axis, and is centered around `center`; i.e., if center is $(center_x, center_y, center_z)$, the cylinder's z-extent is from $center_z - \frac{height}{2}$ to $center_z + \frac{height}{2}$.

Once this is done, try implementing a cylinder that is arbitrarily oriented. You will need to add two parameters, `rotX` and `rotY`, to specify the cylinder's rotation about the $x$ and $y$ axes. Note that this will require changing the coordinate system of the incoming ray within the `intersect` method to account for these rotations. After the coordinate system change, the math used to intersect the ray with the cylinder should be the same as its axis-aligned variant. Finally, build a test scene with a cylinder object.

# 8 Appendix C: Framework

This section describes the framework code that comes with the assignment. You do not need to spend time trying to understand it and it is not essential to read this to get started on the assignment. Instead, you can reference it as needed.

The framework for this assignment includes a simple main program, some utility classes for vector math, a parser for the input file format, and stubs for the classes that are required by the parser.

## 8.1 `BasicRayTracer`

This class holds the entry point for the program. The `main` method is provided, so that your program will have an command-line interface compatible with ours. It treats each command line argument as the name of an input file, which it parses, renders an image, and writes the image to a PNG file. The method `BasicRayTracer.renderImage` is called to do the actual rendering.

## 8.2 `Image`

This class contains an array of `floats` and the requisite code to get and set pixels and to output the image to a PNG file.

## 8.3 The `ray.math` package

This package contains classes to represent 2D and 3D points and vectors, as well as RGB colors. They support all the standard vector arithmetic operations you're likely to need, including dot and cross products for vectors and addition and scaling for colors.

### 8.4 `Workspace`

Each pixel requires a few data elements to compute the color. Allocating them for each new pixel can be quite costly. Further, each pixel is completely independent so the data space can be reused each pixel without causing any computation errors. By placing all the data elements into this short class, this working space can be passed to the pixel shading method each time and avoid the unnecessary allocation.

### 8.5 `Parser`

The `Parser` class contains a simple 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 input

```
<scene>
    <surface type="Sphere">
        <shader type="Lambertian">
            <diffuseColor>0 0 1</diffuseColor>
        </shader>
        <center>1 2 3</center>
        <radius>4</radius>
    </surface>
</scene>
```

results in the following construction sequence:

1. Create the scene.
2. Create an object of class `Sphere` and add it to the scene by calling `Scene.addSurface`. This is OK because `Sphere` extends the `Surface` class.
3. Create an object of class `Lambertian` and add it to the sphere using `Sphere.setShader`. This is OK because `Lambertian` extends the `Shader` class.
4. Call `setDiffuseColor(new Color(0, 0, 1))` on the shader.
5. Call `setCenter(new Point3D(1, 2, 3))` on the sphere.
6. Call `setRadius(4)` on the sphere.

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 header comment of the `Parser` class.

The practical result of all this is that your ray tracer is handed an object of class `Scene` that contains objects that are in one-to-one correspondence with the elements in the input file. You shouldn't need to change the parser in any way.

The input file for your ray tracer is in XML. An XML file contains sequences of nested *elements* that are delimited by HTML-like angle-bracket tags. For instance, the XML code:

```
<scene>
    <camera type="PerspectiveCamera"></camera>
    <surface type=Sphere>
        <center>1.0 2.0 3.0</center>
    </surface>
</scene>
```

contains four elements. One is a `scene` element that contains two others, called `camera` and `surface`. The `camera` element has an *attribute* named `type` that has the value `PerspectiveCamera`. The `surface` element also has an *attribute* named `type` that has the value `Sphere`. It also contains a `center` element that contains the text "1.0 2.0 3.0", which in this context would be interpreted as the 3D point $(1, 2, 3)$.

An input file for the ray tracer always contains one `scene` element, which is allowed to contain tags of the following types:

- `surface`: This element describes a geometric object. It must have an attribute `type` which is the name of the actual class of the surface. It can contain a `shader` element to set the shader, and also geometric parameters depending on its type:
  - for sphere: `center`, containing a 3D point, and `radius`, containing a real number.
  - for box: `minPt` and `maxPt`.
  - for cylinder (see Appendix B): `center`, containing a 3D point, `radius` and `height`, each containing a real number.
- `camera`: This element describes the camera. It must have an attribute `type` which must take either the value `PerspectiveCamera` or `ParallelCamera`. The allowed subelements have been listed in Section 3.1.
- `light`: This element describes a light. It contains the 3D point `position` and the RGB color `color`.
- `shader`: This element describes how a surface should be shaded. It must have an attribute `type` with value `Lambertian` or `Phong`.

If the same object needs to be referenced in several places, for instance when you want to use one shader for many surfaces, you can use the attribute `name` to give it a name, then later include a reference to it by using the attribute `ref`. For instance:

```
<shader type="Lambertian" name="gray">
    <diffuseColor>0.5 0.5 0.5</diffuseColor>
</shader>
<surface type="Sphere">
    <center>0 0 0</center>
    <shader ref="gray"/>
```

```
</surface>
<surface type="Sphere">
    <center>5 0 0</center>
    <shader ref="gray"/>
</surface>
```

applies the same shader to two spheres.

Really, the file format is very simple and from the examples we provide you should have no trouble constructing any scene you want.