Assignment 3 COMP 557 Winter 2015
Ray Tracer

Posted: Wednesday, March 18th
Due: Sunday, April 5 at 23:59

General Information

- This assignment is worth 12 % of your final course grade. It is out of 120 points.
- Charles and Shayan will handle the office hours and grading. Office hours will be posted on the public web page.
- Use the myCourses discussion board (Assignments/A3) for clarification questions.
- Do not change any of the given code. Add code only where instructed.
- Submit the entire directory as a zip or tar file to the myCourses Assignment/A3 folder. The file should be named FirstnameLastname.zip (or .tar) and should unpack into a directory with that same name. If you have any issues that you wish the TAs to be aware of when grading, then include them as a separate comment with your submission.
- **Late assignments** will be accepted up to only 3 days late and will be penalized by *1 point per day on your final course grade*. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. Note that the assignment has 120 points, so the late penalty is 10 points per day.

Introduction

In this assignment, you will implement the main elements of a simple ray tracer. The lighting model used is Blinn-Phong with shadows. As discussed in the lectures, Blinn-Phong includes materials that have a specular, diffuse, and an ambient term. Mirror surfaces are not included in this assignment, unlike the ray tracing method discussed in the lecture.

Each scene is encoded as an XML file. These files can be found in the `scenes/` folder. To render a scene, run `A3App.py` with the XML filename as command-line argument (see the comments in `A3App.py` for details). `A3App.py` calls `SceneParser` with the XML filename (see `A3App.main()`). `SceneParser` parses the scene file and creates all the necessary objects and adds them to the `Scene` object. Once the `Scene` class is initialized, `Scene.renderScene()` method is called to render the scene. The `Scene.render` variable, which is an instance of `Render` class, is responsible for storing and displaying the rendered images. The rendered images are stored in the `images/` folder (the `Render` class will create this folder automatically if needed).

There are two remaining parts to this document. First, we specify the specific requirements of the assignment. You should begin by scanning through them to familiarize yourself with what you must eventually do. Then, carefully read through the **Getting Started** section which follows.

Requirements

There are two sets of requirements. These are grouped by the two files in which you will be adding code, namely `Intersectable.py` and `Scene.py`.

1. **Finding Ray Intersections with Objects (total 60 points)**

   Fill in the missing code for the `intersect()` method of the following classes, which are each defined in the file `Intersectable.py`.

   (a) `Sphere` **(10 points)**

   We suggest you test this one with `sphere.xml` which includes a single sphere at the origin. You should also run `TestSphereIntersection.py` to test different intersection scenarios.

   (b) `Plane` **(20 points)**

   Note that the plane is assumed to contain the origin. For a more general plane, you would need to apply a translation. Run `TestPlaneIntersection.py` to test only the intersection part of your implementation.

   A `Plane` may have either uniform material or it may have square unit "tiles" with two materials which form a checkerboard texture. Implement the checkerboard texture, assuming the plane is $y = 0$. (To make a checkerboard wall, you would have to use a $y = 0$ `Plane` inside a `SceneNode` with an appropriate transformation.)

   (c) `Box` **(10 points)**

   A `Box` is an axis aligned rectangular solid, defined by the min and max corners. After implementing the ray-box intersection, run `TestBoxIntersection.py` to test your implementation. Once shading is implemented, you can render `box.xml`, which uses different colored lights in different axis directions.

Note that this method requires more code than the previous two, since you need to test the six faces of the cube. We suggest you solve the problem for one face, and then copy/paste/edit the solutions for the other five faces (rather than writing a helper method).

(d) `SceneNode` **(20 points)**

Each `SceneNode` object has a transform matrix $\mathbf{M}$ that maps from the object coordinate system to a scene coordinate system. As defined in the `SceneNode` constructor, this transformation first scales, then rotates, and then translates.

The intersection of a ray with a node must be performed in the node's coordinate system. Use the inverse of $\mathbf{M}$ to transform a ray into the node's coordinate system. After you find the intersection, transform the intersection point and the normal back into the coordinate system in which the ray was originally defined.

Use `TestSceneNodeIntersection.py` to unit-test the intersection component of your code.

A few tips:

- Be careful with how you transform the normal.
- Be careful with how the distance to the intersection point is computed. This distance should be meaningful in the scene coordinate system where distances to different objects must be compared.
- Check the correctness of your code by using one of the ellipsoid scenes. Ellipsoids involve non-uniform scaling and this can cause problems with surface normals when there are errors in the code.
- The intersection test must be performed on all of the children of a scene node. Since a child node can be a `SceneNode` node, this may result in recursive calls. The `boxes.xml` scene has a hierarchy of nodes and thus can be used to test your solution code.

2. Rendering (total 60 points)

Fill in the missing code for following methods which are defined in `Scene.py`. See the comments within each method for further details.

(a) `create_ray()` **(20 points)**

Create a ray to cast into the scene.

(b) `get_nearest_object_intersection()` **(10 points)**

Find the nearest visible point on ray and return useful properties of this intersection point.

(c) `get_visible_lights()` **(10 points)**

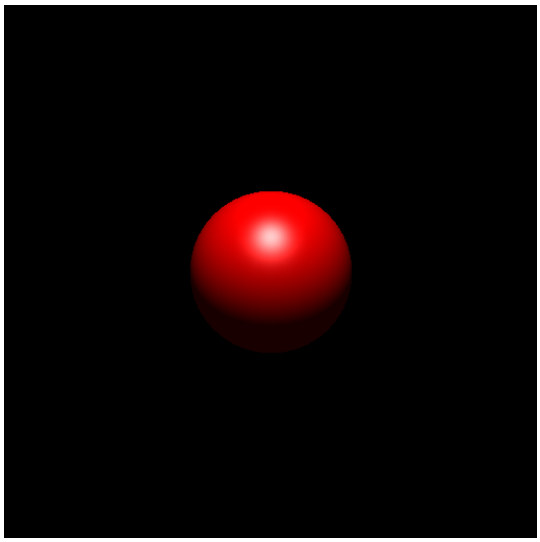
Find the list of lights that is visible from a scene point.

One good scene to use for testing here is `scene4_rot`.

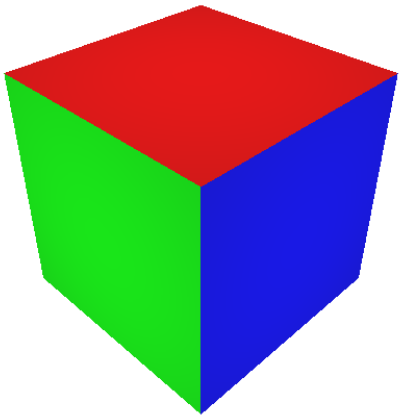
(d) `blinn_phong_shading_per_light()` **(20 points)**

Compute the diffuse and specular components. This method is called by the method `blinn_phong_shading()` which loops over the visible lights.

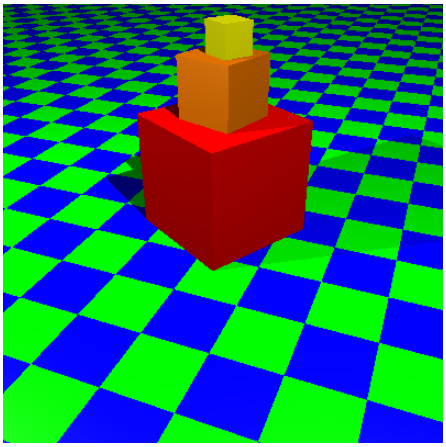
Here are some examples of rendered images which you can find in `images_sol/` .



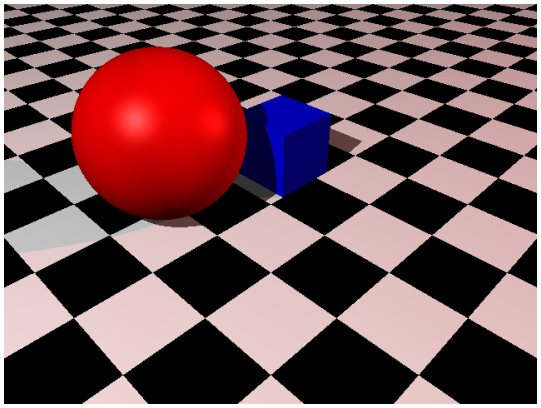
`sphere`



`box`



`boxes`



`scene4_rot`

Getting started

Here we outline some initial steps to get you started. You only have to change the contents of `Scene.py` and `Intersectable.py` files, and the comments in these files will guide you through the process of implementing each part. In `HelperClasses.py` you can find `Camera`, `Render`, `Material` and `Light` classes. In `Ray.py` you have `Ray` and `IntersectionResult` classes. It will be helpful if you scan through these files and the comments before starting. Additionally, there are several tester classes. The name of these files start with `Test`. You can run all the scenes with a fixed resolution using `TestRenderAllScenes`. Read the comments in these files for more information.

Each scene is encoded as an XML file. The starter code includes `SceneParser.py` for parsing the XML scene files. Each XML file is organized as sequence of named materials, lights, cameras, nodes (`SceneNode`), and geometry (`Plane`, `Sphere`, or `Box`). The main scene is a root node in the XML. XML nodes can be organized into a hierarchy as you will see in the question that involves the `SceneNode` class. See the comments at the top of `SceneParser.py` for the scene file requirements. Look at the provided examples in `scenes/` folder to get a better idea of how the scene description files are organized. Feel free to make your own scenes and share them on the discussion board.

To see some examples, look into the `images_sol/` folder and choose a scene and then use the corresponding xml file that is in `scenes` folder.

Here are some suggested steps for getting started:

Set all pixels to background color

In the `renderScene()` method of the `Scene.py` file, you will implement the `create_ray()` method by generating a ray for each pixel of the screen. Each ray starts at the camera position and passes through the point in space that corresponds to the image pixel coordinate (row, col) on near plane of the camera. Implement `create_ray` method so that it creates a ray that goes through the given pixel. The ray origin is the camera's eye position i.e. `Camera.pointFrom`. The camera points to the point in space defined by `Camera.pointTo`. You need to use the camera parameters found in the `Camera` class in `HelperClasses.py` to determine the origin and direction of the ray that would go through the given pixel. Note that `pixel` is a python list containing [col, row] or [x, y] image coordinate. The ray needs to be in the world coordinate system in which the camera coordinate is specified.

To get started you can just consider the camera to be at the origin and looking at the negative z-axis. Later on you can implement a more general version where the camera is at some arbitrary location in space and looking at some arbitrary direction.

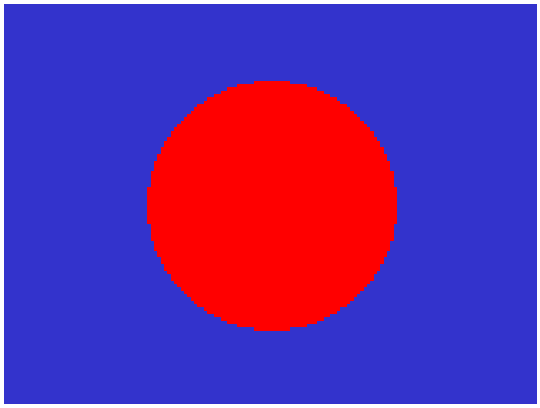
There's a basic scene description included in `Scene.py` which will be used whenever `Scene.py` is executed. The scene contains a red sphere in front of a blue background. You should get a blue background when running `Scene.py` as shown below.

Red sphere on blue background

The goal of this step is to get a simple red sphere on a blue background without any shading. Implement the `intersect()` method of Sphere in `Intersectable.py` file so that it returns a basic `IntersectionResult` object when a ray intersects with the sphere. Then call the `get_nearest_object_intersection()` method. In this method you are supposed to loop through intersectable objects in your scene and find the nearest intersection.

`Scene.surfaces` contain a list of intersectable objects. A surface can be any type specified in `Intersectable.py`, namely `Sphere`, `Plane`, `Box`, and `SceneNode`. The result of the intersection should be returned in a variable of type `IntersectionResult` which is defined in the `Ray.py` file. An `IntersectionResult` object contains the point of intersection (p), the surface normal (n) at the intersection point, material at the intersection point (material) and distance (t) of the intersection point from the ray origin (i.e. `ray.eyePoint`)

At this point for testing purposes you should set a constant red color to a ray when it intersects with your sphere instead of implementing lighting. If you run the `Scene.py` file you should see an output similar to the figure below.



Still getting the blue image? Add the following after creating your ray. This will go through the center of the sphere and has to intersect.

```
ray = Ray([0, 0, 4], [0, 0, -1])
```

You can test Sphere.intersect using `TestSphereIntersection.py`. It's okay if all the tests do not pass the first time. At least a few should pass.

More complex scenes

Once you're ready to try out the raytracer for a more complex scene, you should run `A3App.py`. From the terminal:

```
python A3App.py ./scenes/scene_filename.xml
```

where `scene_filename` is the scene file you want to use. E.g. `./scenes/sphere_blue_bkgnd.xml` will render a similar red sphere on blue background.

Alternatively, you can change the default scene variable DEFAULT_SCENE_FILE located in `A3App.py` so that you do not have to give it an argument.

Finally...

Ray tracing can be slow! The size of the image that you use when working out solutions will depend on the speed of your computer and your patience. It might be sufficient to use 50×50 images for testing initial solutions. You can change the render size of the scenes in the camera element of the xml files. You can also create your own scenes by making a copy of an xml file and changing it yourself.

Get started early. Good luck, and we hope you have fun.