6.837: Computer Graphics Fall 2016

Programming Assignment 4: Ray Tracing

Due Wednesday, November 2nd at 8:00pm.

In this assignment, you will implement a ray caster and ultimately a recursive ray tracer. As seen in class, a ray caster sends a ray for each pixel and intersects it with all the objects in the scene. Ray tracers start the same way, but recursively send additional rays bouncing off from the object intersections in the scene. Your complete assignment will be able to render several basic primitives (spheres, planes, and triangles) and loaded meshes with the Phong shading model, along with options for creating shadows and reflective surfaces through ray tracing.

The handout is organized as follows:

1. Getting Started
2. Summary of Requirements
3. Starter Code
4. Implementation Notes
5. Test Cases
6. Hints
7. Extra Credit
8. Submission Instructions

1 Getting Started

This assignment differs from previous assignments in that there is no OpenGL, and you will write more code. **Please start as early as possible!** You will start by building a ray caster then expand your solution to build a ray tracer.

The sample solution is included in the starter code under the `sample_solution` folder. You may need to change the permissions of the Linux/Mac executables to be able to run them (type `chmod u+x sample_solution/athena/a4` at the prompt). Run the sample solution `a4` as follows:

```
a4 -input ../data/scene01_plane.txt -output out1.png -size 200 200
```

This will generate an image named `out1.png`. We'll describe the rest of the command-line parameters later. When your program is complete, you will be able to render this scene as well as well as the other test cases given below.

2 Summary of Requirements

This section summarizes the core requirements of this assignment. Let's walk through them.

- **Light sources and Shading (30%)** You will implement point light sources and the Phong reflectance model.
- **Planes, Triangles, and Transform Nodes (40%)** Using object-oriented techniques, you will make your ray tracer flexible and extendable. A generic `Object3D` class will serve as the parent class for all 3D primitives. You'r job is to implement specialized subclasses.
- **Recursive Ray Tracing and Shadows(30%)** After the ray caster works, you will implement ray tracing by making your function call in `Renderer` recursive. You will also determine visibility by casting shadow rays.

3 Starter Code

As always, you can add files or modify any files. You can even start from scratch.

- We have included a command-line argument parser (`ArgParser`) to allow easy access to command-line parameters and flags. We have also included
- `SceneParser` which reads in the text files and associated images/.obj's to parse the specified scene. Several constructors and the `Group::addObject` method you will write are called from the parser. Take a look at one of the text files in the data folder to get an idea of how a scene is specified.
- The `Image` class is used to initialize and edit the RGB values of images. The class also includes functions for saving simple PNG image files. Your final product will take in scene text files, calculate the correct pixel colors, and save it as a PNG, probably in the `Renderer` class.
- We provide you with a `Ray` class and a `Hit` class to manipulate camera rays and their intersection points. A `Ray` is represented by its origin and direction vectors. The `Hit` class stores information about the closest intersection point, normal, the value of the ray parameter t and a pointer to the `Material` of the object at the intersection.
- The `Hit` data structure must be initialized with a very large t value (try `std::numeric_limit<float>()`). It is modified by the intersection computation to store the closest t and the `Material` of intersected object.

4 Implementation Steps

Below is a suggested recipe to follow to get as far as possible, as quickly as possible. The underlined pieces give the main goal of each point. The general flow is as follows. The main function parses the command-line arguments and scene and passes that information into the `Renderer`. There, you will calculate the value of each pixel in the output image by ray casting/tracing depending on the scene's objects, making use of the intersection and shading methods you will write.

1. Examine the abstract `Object3D` class. You cannot create an instance of an abstract class directly, but you can use the abstract class by instantiating one of its child classes. It has a method to calculate if a given ray intersects with the object of interest and it also stores a pointer to its `Material` type. Examine the `Sphere` class which inherits from `Object3D` and implements the `intersect()` method. We have implemented `Sphere` for you, though you will be implementing other subclasses of `Object3D`.

   With the `intersect` routine, we are looking for the closest intersection along a `Ray`, parameterized by t. `tmin` is used to restrict the range of intersection. If an intersection is found such that $t >$ `tmin` and t is less than the value of the intersection currently stored in the `Hit` data structure, `Hit` is updated as necessary. Note that if the new intersection is closer than the previous one, the `Hit` object will need to be modified, and `intersect` must return true. Note that `Hit` is passed by reference, meaning that modifying it within the method will modify the original object passed in.

2. Examine the remaining starter code.

   - In `Renderer::Render()` you are provided with the main "for-each pixel" loop of the ray tracer. `Render()` calls the `traceRay()` method for each camera ray — it is your job to implement `traceRay()`.
   - Look at `PerspectiveCamera`. It is implemented for you, but you should understand what the code does.
   - Before getting started with the next step, take a look at `DirectionalLight`, to get familiar with the interface of the `getIllumination()` method of light sources.

3. Implement a point light source . First, you will need to implement the `PointLight::getIllumination()` method. This function takes a point in space, and returns

   (a) The direction vector from scene point to light source (normalized)

   (b) The illumination intensity (RGB) at this point.

   (c) The distance between scenen point and light source.

   The intensity of a point light positioned with distance d from the scene point x_{surf} is given by

   $$L(x_{\text{surf}}) = \frac{I}{\alpha d^2}$$

   where I is the light source color, and in the denomiator is the distance-squared falloff of point light sources. Sometimes, the (physically correct) inverse-square falloff can be too steep in computer graphics. Thus, we multiply by attenuation factor α, for more artistic control

4. Implement diffuse shading . With the point light source generating illumination values, you will need to start implementing the Phong shading model in `Material::shade`. Given the direction to the light $\mathbf{L}$ and the normal $\mathbf{N}$ we compute the diffuse shading as a clamped dot product:

   $$\text{clamp}(\mathbf{L}, \mathbf{N}) = \begin{cases} \mathbf{L} \cdot \mathbf{N} & \text{if } \mathbf{L} \cdot \mathbf{N} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Combined with diffuse material reflectance k_{diffuse} and light intensity L, the diffuse illumination term is.

$$I_{\text{diffuse}} = \text{clamp}(\mathbf{L} \cdot \mathbf{N}) * L * k_{\text{diffuse}}$$

Each color channel is multiplied independently.

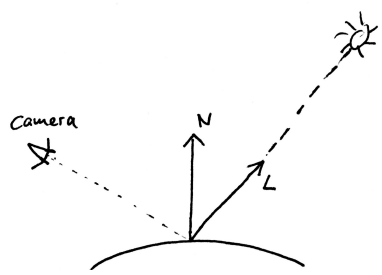


Figure 1: The normal and light vectors for diffuse shading.

5. <u>Implement specular component in the Phong shading model</u>. The specular intensity depends on: shininess s, surface-to-eye direction $\mathbf{E}$, perfect reflection of eye vector $\mathbf{R}$, direction to the light $\mathbf{L}$ and the surface normal $\mathbf{N}$.

   The formula for the specular shading term is

   $$I_{\text{specular}} = \text{clamp}(\mathbf{L}, \mathbf{R})^s * L * k_{specular}$$

   The clamped dot product is similar to the one in the diffuse term. However, now we form it between the reflected eye ray $\mathbf{R}$ and light direction $\mathbf{L}$, which causes the specular highlight to move as the camera moves. Also, we raise the clamped dot product to th power s. Higher shinines s makes the highlight narrower and the surface appear more shiny, smaller s gives the surface a more matte appearance.

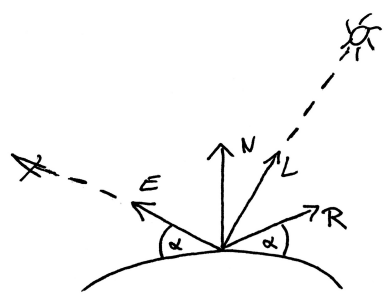


Figure 2: The reflection, normal, and light vectors for specular shading.

6. <u>Combine Diffuse, Specular and Ambient Shading.</u> Our scenes have ambient illumination L_{ambient}, and several light sources that contribute illumination to the object.

   The ambient illumination term is

   $$I_{\text{ambient}} = L_{\text{ambient}} * k_{\text{diffuse}}$$

   .

   To combine ambient illumination and per-light diffuse and specular terms, simply sum them as in

   $$I = I_{\text{ambient}} + \sum_{i \in \text{lights}} I_{\text{diffuse},i} + I_{\text{specular},i}$$

Intensity value I is the final pixel intensity that is written to the frame buffer.

7. Fill in `Plane`, an infinite plane primitive derived from `Object3D`. Use the representation of your choice, but the constructor is assumed to be as in the starter. d is the offset from the origin, meaning that the plane equation is $\mathbf{P} \cdot \mathbf{n} = d$. Implement `intersect`, making sure to update the normal stored by `Hit`, in addition to the intersection distance t and color. You should be able to render Scene 1's spheres and plane correctly now.

8. Fill in `Triangle` which also derives from `Object3D`. The constructor takes 3 vertices, normal for each vertex and a material. Use the method of your choice to implement the ray-triangle intersection. You will need barycentric coordinates to interpolate the normal direction over the surface of the triangle. (To solve a 3x3 linear system, you can formulate it as $Ax = b$ and use `Matrix3f::inverse()` to solve the sytem.)

   With triangle intersection implemented correctly, you should be able to render Scene 2's cube.

9. Fill in `Transform`, another `Object3D` subclass. `Transform` stores a pointer to a child `Object3D` node. It also stores a 4x4 transformation matrix $\mathbf{M}$. This matrix moves descendants of the `Transform` node from local object coordinates to world coordinates.

   For complicated child objects, such as meshes with many vertices, it would be prohibitive to move the entire object into world space whenever we want to trace a ray. It is much cheaper to instead *move the ray from world space into object space*. This is what you'll have to implement.

   One a hit is found, the hit normal is in object coordinates. You'll have to transform the normal from local coordinates back to world coordinates. (Remember, when transforming normal directions, you need to transform by the inverse-normal of the transform matrix).

10. Run your ray caster on Scenes 1-5. Congrats, you've created a ray caster! Now it is time to extend it to a ray tracer to cast secondary rays that account for reflection and shadows so that we may render the shiny bunny in Scene 6 and the arch in Scene 7.

11. Support `-bounces` . `traceRay` will be recursive for specular materials. The maximum recursion depth is passed as a command line argument `-bounces max_bounces`. Play with this argument on the sample solution. Note that even if `max_bounces` $= 0$, we are effectively ray casting (as there is no recursion).

    Implement mirror reflections for reflective materials by sending a ray from the current intersection point into the direction of perfect reflection. This is the same direction as $\mathbf{R}$ in the specular Phong term.

    Trace the secondary ray with a recursive call to `traceRay` using modified recursion depth.

    Add the color seen by the reflected ray times the specular material reflectance to the color computed for the current ray. If the direct illumination (ambient, diffuse, specular) is I_{direct}, the total intensity of direct and indirect illumination is

    $$I_{\text{total}} = I_{\text{direct}} + k_{\text{specular}} * I_{\text{indirect}}.$$

    If a ray didn't hit anything, simply return the background by `SceneParser::getBackgroundColor(dir)`.

    With bounces implemented, you should be able to render the reflective bunny (scene 6).

12. Support `-shadows`. To compute cast shadows, you will send rays from the surface point to each light source. If an intersection is reported, and the intersection is closer than the distance to the light source, the current surface point is in shadow and direct illumination from that light source is ignored. Note that shadow rays must be sent to all light sources.

    Recall that you must displace the ray origin slightly away from the surface, or equivalently set `tmin` to some ϵ. Refer to the document http://www.cs.utah.edu/~shirley/books/fcg2/rt.pdf for more details.

    Congratulations, with shadows and reflections, you can now correctly render scene 7.

5 Test Cases

Your assignment will be graded by running a script that is provided in the starter distribution. below. Make sure your ray caster produces equivalent output. During development, you can test your renderer with the following commands. The scripts in the starter distribution show how to use other parameters, such as `-depth` and `-normals`.

Ray Casting

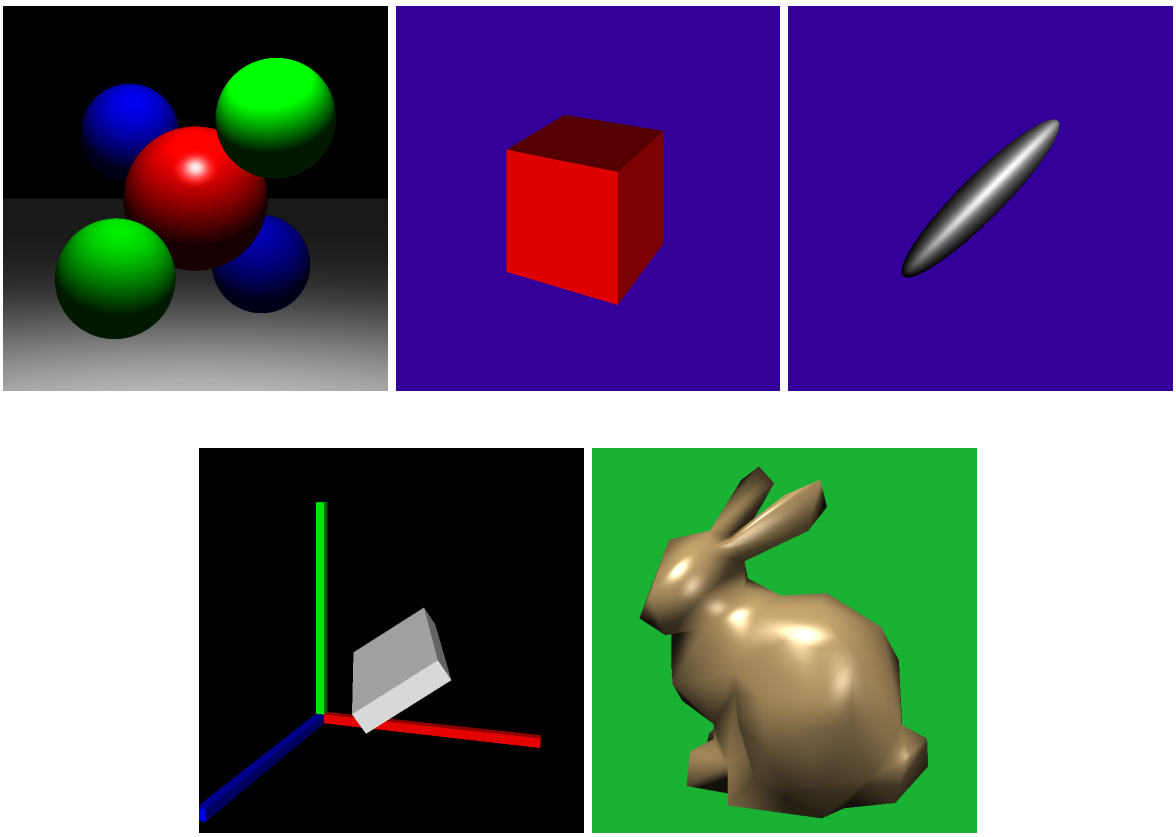
```
./a4 -input ../data/scene01_plane.txt -output01.png -size 200 200

./a4 -input ../data/scene02_cube.txt -output02.png -size 200 200

./a4 -input ../data/scene03_sphere.txt -output03.png -size 200 200

./a4 -input ../data/scene04_axes.txt -output04.png -size 200 200

./a4 -input ../data/scene05_bunny_200.txt -output05.png -size 200 200
```

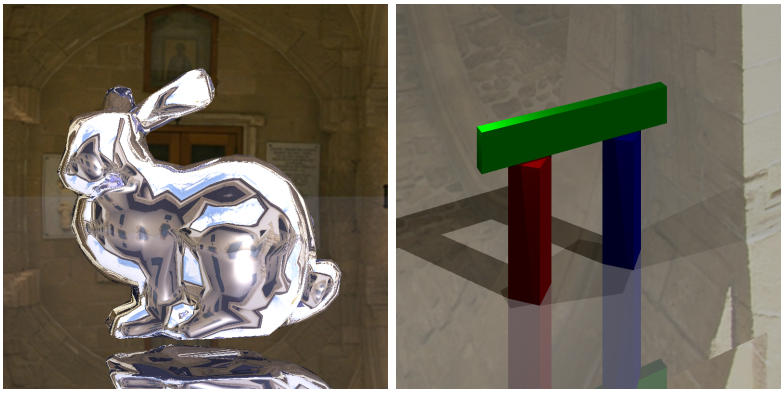


Ray Tracing

For ray tracing, we must enable recursive bounces. Four bounces are usually enough. Zero bounces means just the camera rays, with no recursion at all.

```
./a4 -input ../data/scene06_bunny_1k.txt -output06.png -size 300 300 -bounces 4

./a4 -input ../data/scene07_arch.txt -output07.png -size 300 300 -shadows -bounces 4
```



6 Hints

- Incremental debugging. Implement and test one primitive at a time. Test one shading add-on at a time.
- Use a small image size for faster debugging. 200×200 pixels is usually enough to realize that something might be wrong. Use higher resolution (e.g. 800x800) to debug details of your geometry and shading (we will grade based on high resolution images).
- As usual, don't hesitate to print as much information as needed for debugging, such as the direction vector of the rays, the hit values, etc.
- Use `assert()` to check function preconditions, array indices, etc. See `cassert`.
- To avoid a segmentation fault, make sure you don't try to access samples in pixels beyond the image width and height. Pixels on the boundary will have a cropped support area.

7 Extra Credit

Most of these extensions require that you modify the parser to take into account the extra specification required by your technique. Make sure that you create (and turn in) appropriate input scenes to show off your extension.

7.1 Easy

- Add anti-aliasing to your ray tracer through supersampling and filtering to alleviate jaggies. In the solution, we use jiterred sampling and Gaussian blur with images supersampled at a factor of 3. Add flags `-jitter -filter` to view the result.
- Add simple fog to your ray tracer by attenuating rays according to their length. Allow the color of the fog to be specified by the user in the scene file.
- Add support for refraction rays and refractive materials.
- Add other types of simple primitives to your ray tracer, and extend the file format and parser accordingly. For instance, how about a cylinder or cone? These can make your scenes much more interesting.

- Add a new oblique camera type (or some other weird camera). In a standard camera, the projection window is centered on the z-axis of the camera. By sliding this projection window around, you can get some cool effects.
- Add more interesting lights to your scenes, e.g. a spotlight with angular falloff. You can also give more artistic control over the point light source, by adding constant or linear falloff terms. See https://developer.valvesoftware.com/wiki/Constant-Linear-Quadratic_Falloff

7.2 Medium

- Implement a torus or higher order implicit surfaces by solving for t with a numerical root finder.
- Bump mapping: look up the normals for your surface in a height field image or an normal map. This needs the derivation of a tangent frame. There many such free images and models online.
- Load or create more interesting complex scenes. You can download more models and scenes that are freely available online.
- Bloom (light glow) and High-Dynamic Range rendering: render multiple passes and do some blurring. See wikipedia article on Bloom and GPU Gems article for more info.
- Add area light sources and Monte-Carlo integration of soft shadows.
- Render glossy using Monte-Carlo integration.
- Render interesting BRDF such as milled surface with anisotropic reflectance.
- Distribution ray tracing of indirect lighting (very slow). Cast tons of random secondary rays to sample the hemisphere around the visible point. It is advised to stop after one bounce. Sample uniform or according to the cosine term (careful, it's not trivial to sample the hemisphere uniformly).
- Uniform Grids. Create a 3D grid and "rasterize" your object into it. Then, you march each ray through the grid stopping only when you hit an occupied voxel. Difficult to debug.
- Simulate dispersion (and rainbows). The rainbow is difficult, as is the Newton prism demo.
- Make a little animation (10 sec at 24fps will suffice). E.g, if you implemented depth of field, show what happens when you change camera focal distance. Move lights and objects around.
- Add motion blur to moving objects in your animation.

7.3 Hard

- Depth of field blurring. The camera can focus on some distance and objects out of focus are blurred depending on how far it is from the focal plane. It doesn't have to be optically correct, but it needs to be visually pleasing.
- Photon mapping with kd-tree acceleration to render caustics.
- Irradiance caching.
- Subsurface scattering. Render some milk, jade etc.
- Path tracing with importance sampling, path termination with Russian Roulette, etc.

- Raytracing through a volume. Given a regular grid encoding the density of a participating medium such as fog, step through the grid to simulate attenuation due to fog. Send rays towards the light source and take into account shadowing by other objects as well as attenuation due to the medium. This will give you nice shafts of light.
- Animate some water pouring into a tank or smoke rising.

8 Submission Instructions

You are to write a `README.txt` that answers the following questions:

- If you don't use the standard CMake build system: How do you compile your code? Provide instructions for Athena Linux.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.
- Got any comments about this assignment that you'd like to share?

Submit your assignment on Stellar. Please submit a single archive (`.zip` or `.tar.gz`) containing:

- All source code.
- A compiled executable named `a4`.
- Any additional files that are necessary.
- The `README` file.