Programming Assignment 4: Ray Tracing

MIT 6.837 Computer Graphics

Fall 2022

Due November 16 at 8pm, Boston Time

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, including shadows and reflective surfaces using ray tracing.

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 distribution. Look at the `sample_solution` directory for Linux, Mac, and Windows binaries. You may need to change the file mask on the Linux/Mac binaries, e.g., by executing the following command in your terminal:

```
chmod a+x sample_solution/linux/assignment4
```

To run the sample solution, execute

```
sample_solution/linux/assignment4 -input scene01_plane.txt -output out1.png -size 200 200
```

This will parse the scene defined in `assets/assignment4/scene01_plane.txt`, render it, and generate a 200×200 pixel image named `out1.png`. When your program is complete, you will be able to render this scene as well as the other test cases described below.

The relevant starter code for this assignment is in the `assignment_code/assignment4` directory. To build the starter code, follow the same steps as in previous assignments. For instance, on MacOS or Linux, execute the following:

```
mkdir build
cd build
cmake ..
make
```

If you run the starter code now, a window of an empty scene will pop up.

2 Summary of Requirements

This section summarizes the core requirements of this assignment.

- **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 `HittableBase` class will serve as the parent class for all 3D primitives. Your 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 `Tracer` recursive. You will also determine visibility by casting shadow rays.

3 Starter Code

As before, while you are encouraged to build on the the provided starter code, you are not required to do so. We have provided you with the following classes:

- `ArgParser` allows easy access to command-line parameters and flags.
- `SceneParser` reads in the text files and associated images/.obj's to parse the specified scene. Several constructors 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.
- `Image` is used to initialize and manipulate 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 `Tracer` class.
- `Ray` and `HitRecord` are used to manipulate camera rays and their intersection points. A `Ray` is represented by its origin and direction vectors. A `HitRecord` object stores information about the normal and ray parameter t at the closest intersection point. `HitRecord` must be initialized with a very large t value (try `std::numeric_limits<float>::max()`) and is modified by the intersection computation.

4 Implementation Steps

Below is a suggested recipe to follow to get as far as possible, as quickly as possible. The general flow is as follows. The main function parses the command-line arguments and the scene and creates the appropriate `Tracer`. 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 `HittableBase` class (found in `hittable/`) . You cannot create an instance of an abstract class directly, but you can use the abstract class by inheriting it and implementing all the pure virtual member functions. It has a method to calculate if a given ray intersects with the object of interest.

   Examine the `Sphere` class, which inherits from `HittableBase` and implements the `Intersect()` method. We have implemented `Sphere` for you, though you will be implementing other subclasses of `HittableBase`.

   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 `HitRecord` data structure, the `HitRecord` object is updated, and `Intersect` must return true. Note that the method assumes that the ray is in the *object's* coordinate frame, and so, the record should contain geometry also in this coordinate frame.

2. Implement `Tracer`.
   - In `Tracer::Render()` you are provided with the main "for-each pixel" loop of the ray tracer. `Render()` calls the `TraceRay()` method for each camera ray—you must implement `TraceRay()`.

- For each `TracingComponent`, recall that you can access the local-to-world transformation matrix of its associated transform by `p->GetNodePtr()->GetTransform().GetLocalToWorldMatrix()`.
  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.*
  Once a hit is found, the hit normal is in object space. You'll have to transform the normal from local back to world coordinates. **Remember, when transforming normal directions, you must transform by the inverse-transpose of the transform matrix.**
- Look at `PerspectiveCamera`. It is implemented for you, but you should understand the code.

3. Implement a point light source. First, you will need to implement the missing code for point lights in `Illuminator::GetIllumination()` (the implementation for directional light is provided). This method takes a point in space as well as a `LightComponent` object (the lights are contained in `light_components_` of a `Tracer` object) and sets:

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

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

   (c) The distance between hit 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 denominator 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 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 `Tracer`. Given the direction to the light $\mathbf{L}$ and the normal $\mathbf{N}$, compute the diffuse shading as a clamped dot product:

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

   Given diffuse material reflectance k_{diffuse} and light intensity L, the diffuse illumination term is:

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

   Each color channel is multiplied independently.

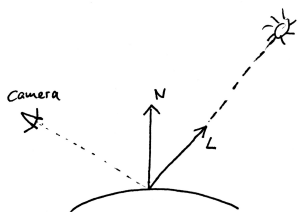


Figure 1: The normal and light vectors for diffuse shading.

5. Implement specular component in the Phong shading model. 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 specular shading is $I_{\text{specular}} = \text{clamp}(\mathbf{L}, \mathbf{R})^s \cdot L \cdot k_{\text{specular}}$.

   Here, we take the clamped dot product 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

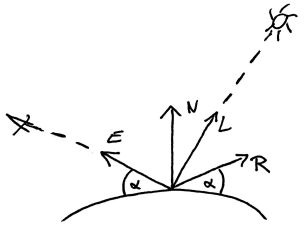


Figure 2: The reflection, normal, and light vectors for specular shading.

product to the sth power. Higher shininess s makes the highlight narrower and the surface appear shinier; smaller s gives the surface a more matte appearance.

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

   The ambient illumination term is $I_{\text{ambient}} = L_{\text{ambient}} \cdot k_{\text{diffuse}}$.

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

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

   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 `HittableBase`. 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 the `HitRecord` object in addition to the intersection distance t.

   You should be able to render Scene 1's spheres and plane correctly now.

8. Fill in `Triangle`, which also derives from `HittableBase`. The constructor takes three vertices and a normal for each vertex. 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 3×3 linear system, you can write it as $Ax = b$ and use `glm::inverse()`.)

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

9. 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.

10. Support `-bounces`. `Tracer::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 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. Note that you will need to move the origin of this new ray by ε towards its direction to avoid detecting the intersection that was previously found.

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

    If a ray didn't hit anything, simply return the background using `GetBackgroundColor(dir)` (this method already implemented for you)

    With bounces implemented, you can render the reflective bunny (Scene 6).

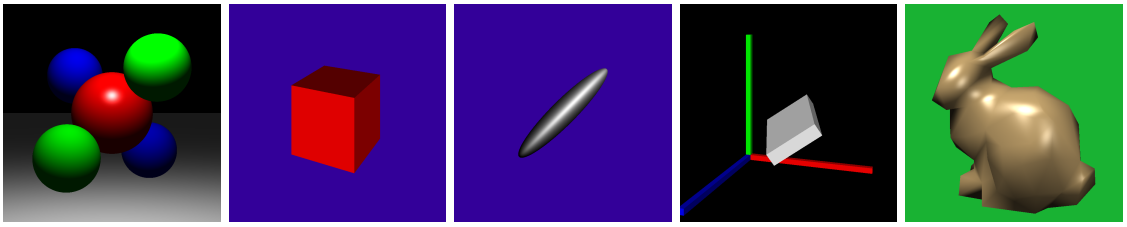
11. Support `-shadows`. To compute cast shadows, you will send rays from the surface point to each light source. If an intersection is reported, and the intersection is closer than the distance to the light source, the current surface point is in shadow, and direct illumination from that light source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

5 Test Cases

During development, you can test your renderer with the following commands. Make sure your ray caster produces equivalent outputs. Your rendered images from these test cases should be submitted along with your code. See Section 8 for more details.

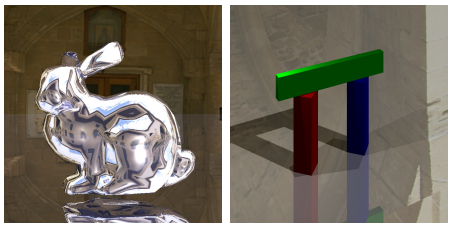
Ray Casting

```
./assignment4 -input scene01_plane.txt -output 01.png -size 200 200
./assignment4 -input scene02_cube.txt -output 02.png -size 200 200
./assignment4 -input scene03_sphere.txt -output 03.png -size 200 200
./assignment4 -input scene04_axes.txt -output 04.png -size 200 200
./assignment4 -input scene05_bunny_200.txt -output 05.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.

```
./assignment4 -input scene06_bunny_1k.txt -output 06.png -size 300 300 -bounces 4
./assignment4 -input scene07_arch.txt -output 07.png -size 300 300 -shadows -bounces 4
```



6 Hints

- 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. 800×800) to debug details of your geometry and shading. (We will grade based on 800 × 800 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.

- 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.
- For debugging purposes, it may help to output and visualize the depth or normal maps for your renderings, similar to how you output the actual rendered image.

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.
- 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?
- 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[1].

7.2 Medium

- Implement a torus or a 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 a normal map. This needs the derivation of a tangent frame. There are 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[2].
- 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.

[1] https://developer.valvesoftware.com/wiki/Constant-Linear-Quadratic_Falloff
[2] https://en.wikipedia.org/wiki/Bloom_(shader_effect)

- 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.
- 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.
- Animate water pouring into a tank or smoke rising.

8 Submission Instructions

You are to include a `README.txt` file or PDF report that answers the following questions:

- How do you compile and run your code? Specify which OS you have tested on.
- 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. In particular, mention if you borrowed the model(s) used as your artifact from somewhere.
- 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 how you did it.
- Do you have any comments about this assignment that you'd like to share? We know this was a tough one, but did you learn a lot from it? Or was it overwhelming?

You should submit your entire project folder including your **source code** and **executable (at the root directory of your project)**, but **excluding `external/` and `build/`** since they take too much space. **Make sure your code can be built successfully, since we will compile it from scratch for grading.**

If you are on Windows, make sure the paths in your `#include "..."` pre-processing commands in `cpp/hpp` files use "/" instead of "\" as file separators. The later will not work on other operating systems.

To sum up, your submission should be your entire project folder, plus:

- The `README.txt` file or PDF report answering the questions above. Leave this file at the root directory of the project folder.

- Images (**resolution 800×800**) from seven test cases (`01-07.png`) at the root folder.
- **As always, TAs will compile and run your code on every test case on their computers. Please make sure your code generates corresponding images when command line arguments are used.** For example,

```
./assignment4 -input scene06_bunny_1k.txt -output 06.png -size 800 800 -bounces 4
```

  should create a 800 × 800 `06.png` file with a bunny at the current working folder.
- If you implemented extra credits, please attach the images and corresponding commands to generate the images using your code.
- The executable file at the root directory of the project folder.

Please compress your project folder into a `.zip` file and submit using Canvas.
We will follow the late day policy explained in Lecture 1.