ECSE 446/546: Realistic/Advanced Image Synthesis

Assignment 2: Point Lights & Shadows

Due Sunday, October 6nd, 2019 at 11:59pm EST on myCourses

10% (ECSE 446 / COMP 598) *or* **5%** (ECSE 546 / COMP 599)

Contents

You will implement two basic reflectance/material models and a simple shading method, both in an offline and interactive context. First, you will implement an offline lighting "integrator" but — at least for the time being — this *integrator* will not actually perform any numerical integration. In the future, however, more complex light transport shading algorithms will actually require numerical integration routines. In addition to your offline integrator, you will implement an interactive shader.

Your shading algorithms will generate images with direct illumination from a point light source. As such, parts of a scene that are not directly lit by the emitter should appear completely dark. Moreover, since you will only be implementing an idealized emitter model in this assigment, the shadows you generate will have "hard" boundaries — i.e., they will not have any penumbra regions. Put differently, every point in the scene is either completely lit or completely shadowed.

Before starting the assignment, make sure that your project is synced up with the online Git repo by running: `git stash save && git pull && git stash pop`. See the TinyRender docs for more information. The base code contains `TODO(A2)` comments; it may be useful to perform a search for these using your IDE.

1 Offline rendering (*50 pts*)

Thread-safe sampler

Before you start, you must modify the random sampler in your rendering loop and supply one sampler *per thread*. In A1, you were asked to instantiate the sampler *before* the rendering loop. However, this can cause visual artifacts in your rendered images the random number generator used in TinyRender is not thread-safe. Simply move the declaration after the first for-loop and offset the seed with the pixel row index:

```
#ifdef NDEBUG // Running in release mode - Use threads
   ThreadPool::ParallelFor(0, scene.config.height, [&] (int y) {
#else // Running in debug mode - Don't use threads
    ThreadPool::SequentialFor(0, scene.config.height, [&](int y) {
#endif

Sampler sampler(yourStudentID + y);
for (int x = 0; x < scene.config.width; x++) { ... }
```

This ensures that each thread consumes a different sampler and that each student uses a unique sampler to render their images.

1.1 Diffuse reflectance model (*5 pts*)

The file `bsdfs/diffuse.h` contains an almost empty `DiffuseBSDF` structure, along with its constructor. Your first task is to implement the `eval()` function, which currently returns 0:

```
v3f eval(const SurfaceInteraction& i) const override {
    v3f val(0.f);
    // TODO: Implement this
    return val;
}
```

In TinyRender, the diffuse albedo ρ is casted to a constant 2D texture, which will later allow us to support spatially-varying albedos stored in bitmap textures. To retrieve the albedo value at a surface interaction point `i`, you will have to invoke the texture's evaluation function by calling `albedo->eval(worldData, i)`.

Your task is to complete the `DiffuseBSDF::eval()` implementation, following these high-level steps:

1. test whether the incoming and outgoing rays are front-facing to the surface — if not, return black; and,
2. otherwise, return the albedo divided by π and multiplied by the cosine foreshortening factor $\cos\theta_i$.

You can use the `Frame::cosTheta()` utility function on `i.wo` and `i.wi` to perform your front-facing tests. These tests are necessary to avoid returning a nonzero color if a ray hits the backface of a triangle, or if we are evaluating the reflectance model in a light direction underneath the (opaque) surface's tangent plane. Drawing a diagram with pen and paper is a good way to get a sense of what values need to be tested, and how.

All BRDF evaluation functions return the BRDF value **multiplied by the cosine factor** — TinyRender always combines the BRDF and cosine foreshortening terms, by convention. Note that the implementation of the `sample()` and `pdf()` routines can be left as-is, as these will be tasks for a future assignment.

1.2 Phong reflectance model (*10 pts*)

Your second task is to implement the Phong reflectance model. In 1975, Phong introduced a reflectance model that combines diffuse and glossy reflection components. Many more advanced and accurate models exist (*e.g.*, based on microfacet theory), however implementing Phong's model in a physically-based renderer remains a nice way to become familiar with what it takes to add different BRDFs to a renderer.

The normalized Phong BRDF is defined as the sum of a diffuse and a glossy component:

$$\begin{aligned} f_r(\mathbf{x}, \omega_i, \omega_o) &= f_{r,d}(\mathbf{x}, \omega_i, \omega_o) + f_{r,s}(\mathbf{x}, \omega_i, \omega_r) \\ &= \rho_d \frac{1}{\pi} + \rho_s \frac{n+2}{2\pi} \max(0, \cos^n \alpha). \end{aligned} \tag{1}$$

- ρ_d is the *diffuse reflectivity/albedo* — the fraction of the incoming energy that is reflected diffusely (clamped to 1 for energy conservation),
- ρ_s is the *specular reflectivity/albedo* — the fraction of the incoming energy that is reflected specularly,
- n is the *Phong exponent* — higher values yield more mirror-like specular reflection, and
- α is the angle between the perfect specular reflection direction ω_r and the lighting direction.

Similarly to the diffuse reflectance model, you will complete the implementation of the `PhongBSDF::eval()` function in the file `bsdfs/phong.h`. Concretely, you must modify the placeholder implementation according to these steps:

1. test for front-facing incoming and outgoing rays,
2. evaluate the Phong BRDF in Equation (1), and
3. return this value multiplied by the cosine foreshortening factor.

You can compute the perfect specular direction ω_r either using the utility function `PhongBSDF::reflect()` or from scratch. The file `core/platform.h` contains some constant definitions that you will need, e.g., for π and $\frac{1}{2\pi}$. Note that `diffuseReflectance`, `specularReflectance` and `exponent` are also casted to texture types, so you will need to evaluate them at your surface points similarly to how you did in the diffuse case.

You need to additionally multiply each reflectivity/albedo component (ρ_d and ρ_s) by `PhongBSDF::scale` to ensure energy conservation and generate the correct result.

1.3 Unshadowed direct illumination (*20 pts*)

To test your new reflectance models, you will implement two simple shading routines: one that computes unshadowed shading and another that computes shadowed shading. In both cases, you will only support shading from an isotropic point light source.

The integrator abstraction encapsulates offline shading algorithms: the complex shading algorithms we will see later in the course will solve integral equations, however the shading algorithms you implement below are much simpler. Their algorithmic logic will be contained primarily in the `SimpleIntegrator::render()` routine of the `SimpleIntegrator` class — here, you will compute a simplified shading routine. Specifically, simplified shading equation you will implement corresponds to

$$L_o(\mathbf{x}, \omega_o) = \frac{I}{r^2}\, f_r(\mathbf{x}, \omega_i, \omega_o)\ \max(0, \cos\theta_i).$$

Here, the light is defined in terms of its radiant intensity $I = \Phi/4\pi$, where Φ is its power (or radiant flux), and r is the distance between the point light and $\mathbf{x}$.

Start by implementing a simple direct illumination integrator (`integrators/simple.h`) that does not include any occlusion queries (i.e., for shadows) between the shading point and the light. As with your `NormalIntegrator`, begin by intersecting your scene with the camera ray provided (by the rendering system) to your `render` routine. If you find an intersection $\mathbf{x}$, retrieve the intersected shape's BRDF at the hit/shading point, evaluate the incident radiance and compute the outgoing radiance contribution:

1. intersect your ray with the scene geometry — if there's an intersection `i`, then
2. retrieve the light position $\mathbf{p}$ and its intensity I,
3. retrieve the intersected surface's material using `getBSDF(i)`,
4. transform the world-space incoming direction `i.wi` to the surface's local coordinates using the hit point's `frameNs.toLocal()` transform, and
5. evaluate the BRDF and incident lighting locally, setting the `Li` term accordingly before computing the outgoing radiance.

Note that, here, we are performing shading in the **local coordinate frame** at the surface, necessitating the transformation of your incoming direction into the surface's local frame. This is necessary to evaluate the BRDF, as the BRDF routines all assume computation in local coordinates at the BRDF point:

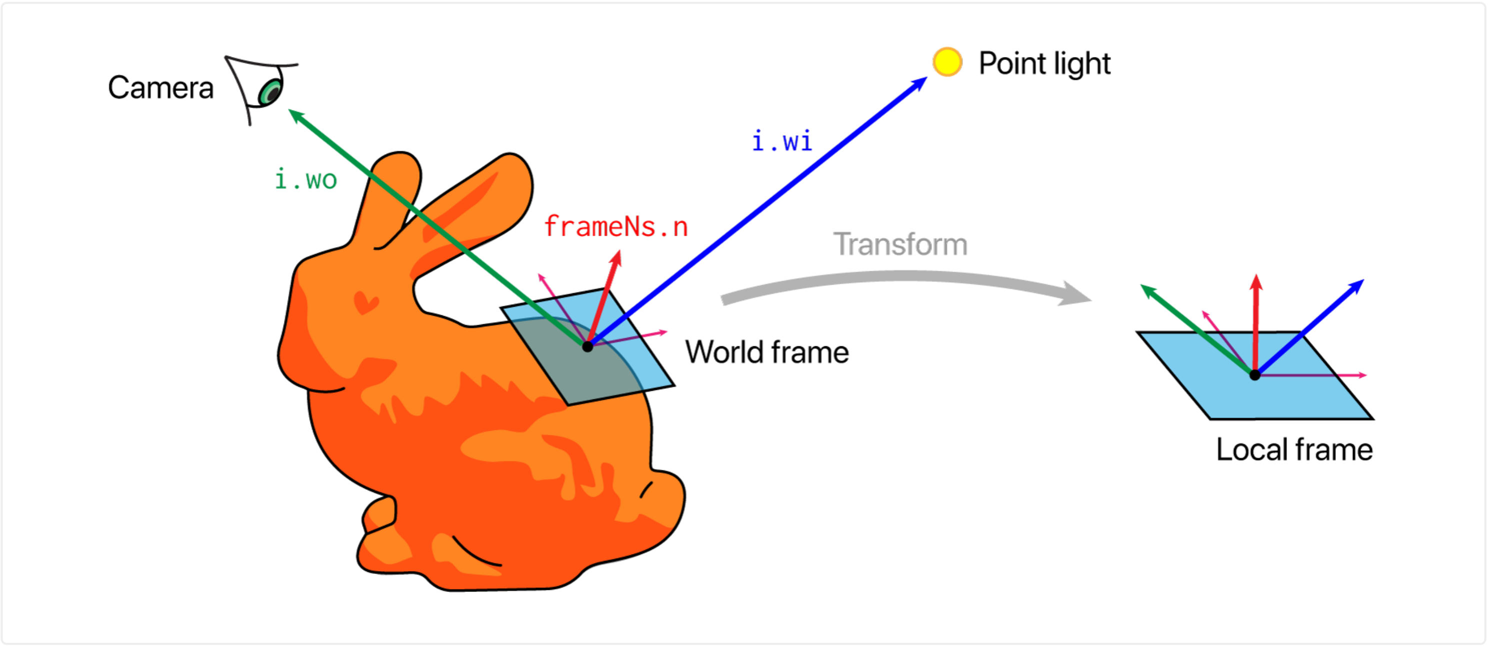

Figure 1: *Mapping from world-space coordinates to the local coordinate frame at a shading point. Sanity check question: if `frameNs.n` is expressed in world-space coordinates, what direction does it map to in local coordinates?*

You will compute the incident radiance L_i directly in your integrator — to avoid dedicating an entire structure to abstract emitters, TinyRender assumes that the light is "attached" to a very small quad mesh at emitter index zero. Hence, to retrieve the position and the intensity of the point light, use:

```
v3f position = scene.getFirstLightPosition();
v3f intensity = scene.getFirstLightIntensity();
```

Note that you need to convert the emitter's radiant intensity to radiant flux by accounting for the squared-distance falloff. The final outgoing radiance **from the emitter** depends on both the radiant intensity I and the squared distance between $\mathbf{x}$ and the light's position $\mathbf{p}$:

$$L_o(\mathbf{p}, \mathbf{p} \to \mathbf{x}) = L_o(\mathbf{p}, -\omega_i) = \frac{I}{\|\mathbf{x} - \mathbf{p}\|^2}.$$

Recall that, in surface light transport, outgoing radiance **from** the emitter equals incident radiance **to** the shade point (*i.e.* $L_o(\mathbf{p}, -\omega_i) = L_i(\mathbf{x}, \omega_i)$), and you are left with everything you need to implement the shading model in this simple integrator. When completed, render the scenes in `data/a1/sphere/tinyrender` — below are reference images for these two offline scenes:

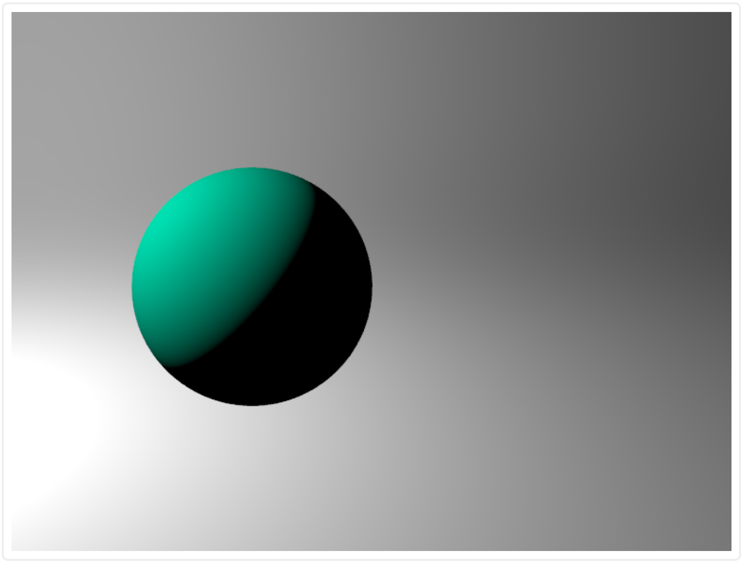
Figure 2: *Unshadowed diffuse sphere scene.*

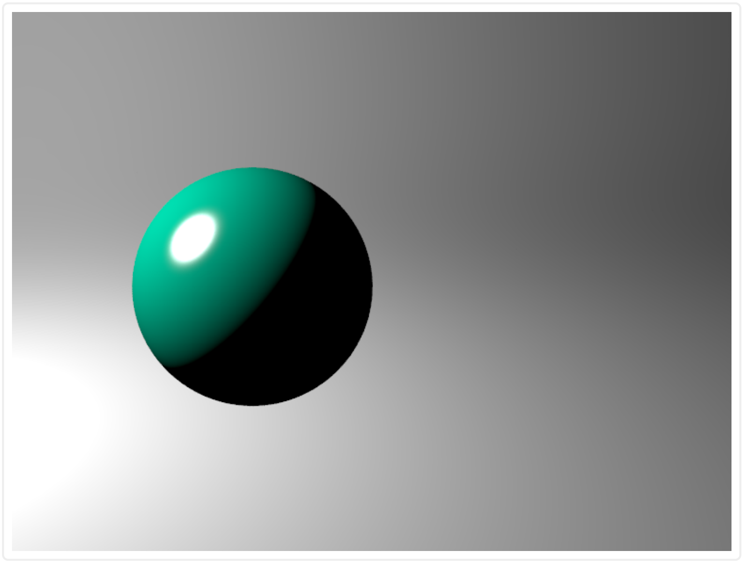
Figure 3: *Unshadowed Phong sphere scene.*

1.4 Direct illumination with hard shadows (*15 pts*)

To compute hard shadows, modify your integrator to evaluate the visibility function $V(\mathbf{x}, \omega_i) = V(\mathbf{x} \leftrightarrow \mathbf{p})$, defined as

$$V(\mathbf{x} \leftrightarrow \mathbf{p}) := \begin{cases} 1, & \mathbf{x} \text{ and } \mathbf{p} \text{ are mutually visible} \\ 0, & \text{otherwise} \end{cases}$$

using a shadow ray query. Intersecting a shadow ray against the scene is generally cheaper than tracing an arbitrary ray, since it suffices to check whether **any** valid intersection exists (rather than having to find the closest one) — this affords an early-out optimization opportunity, which we leverage in our ray tracing implementation. To evaluate visibility, create a shadow ray and set the ray's bounds, e.g. its `max_t` value, accordingly.

You can compare your results with the reference images below.

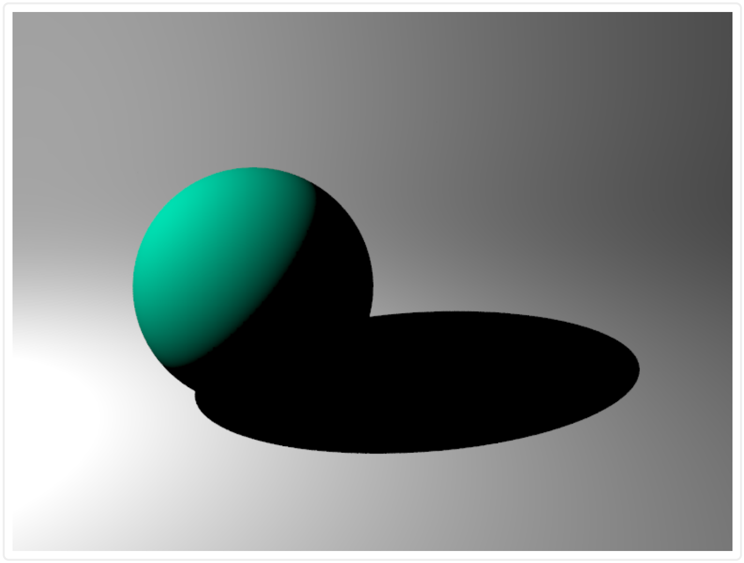
Figure 4: *Diffuse sphere scene with shadows.*

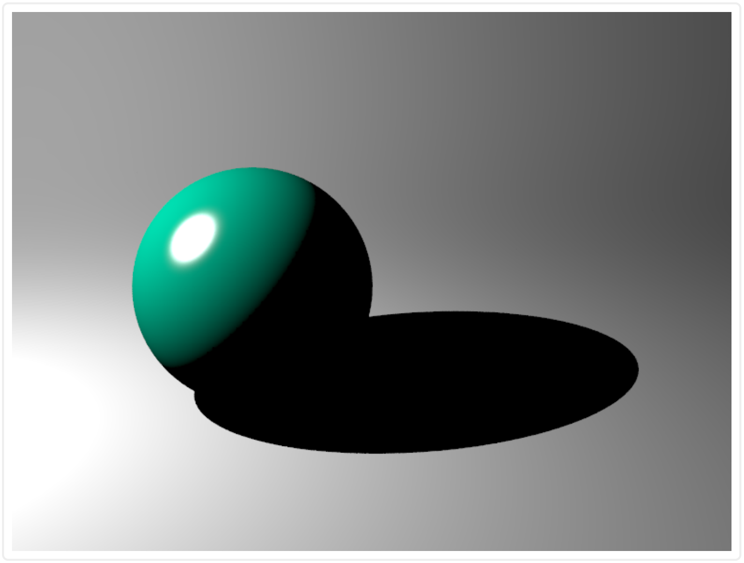
Figure 5: *Phong sphere scene with shadows.*

2 Real-time rendering (*50 pts*)

The real-time/interactive component of this assignment is similar to the offline tasks, and both solutions should yield the same images at 1spp. Once again, your scene will have two types of materials (diffuse and Phong) and a single point light. We ask you to implement fragment shaders for each of these materials; the point light illumination computation will also be implemented inside the shaders. You will be able to share a single vertex shader across the two cases.

While it is possible (and perhaps more efficient) to do all your shading computations in *view space*, this assignment will require that all shading computations be performed in *world space*.

2.1 Render pass (*5 pts*)

Similar to Assignment 1, complete the implementation of `src/renderpasses/simple.h` as follows:

1. pass the light position and intensity using GL uniforms,
2. pass material-specific parameters using GL uniforms, and
3. bind the appropriate shaders (i.e., depending on the material) and draw the object.

Some basic uniforms are already forwarded for you (*e.g.*, model/view/projection matrices). Following the same structure, create GL uniforms for the light and material attributes. Since these are 3D `float` quantities, be careful to use the appropriate `glUniform` routine.

The `GLObject.shaderIdx` variable contains the index of the currently assigned shader (diffuse or Phong):

```
DIFFUSE_SHADER_IDX
PHONG_SHADER_IDX
```

Check to see which shader you are dealing with and create the necessary `glUniform`'s for each case. When complete, do not forget to bind the VAO, draw the triangles and clean the vertex array — just like in the first assignment.

2.2 Vertex shader (*5 pts*)

Start by adding a new vertex shader in `shaders/simple.vs` to your TinyRender codebase. As mentioned above, this vertex shader will be shared across the different fragment shaders.

The vertex shader should be very similiar to your shader from Assignment 1:

Shader uniforms

- Model matrix
- View matrix
- Projection matrix
- Normal matrix

Attributes (`in` variables)

- Vertex position (in *world space*)
- Vertex normal (in *world space*)

What needs to be computed and forwarded to the rasterization unit for interpolation (`out` variables)

- Vertex position (in *world space*)
- Vertex normal (in *world space*)

Implement this vertex shader and don't forget to compute and set `gl_Position` to the vertex position in post-projective screen space coordinates.

2.3 Diffuse fragment shader (*15 pts*)

Next, implement a fragment shader for the diffuse material in a new file named `src/shaders/diffuse.fs`. At a high level, you will implement shading logic that combines the functionality of `DiffuseBRDF::eval()` and your

`SimpleIntegrator` — all in a single, monolithic shader. This is where you get to use the `glUniform`'s you created in Part 2.1.

Shader uniforms

- Camera position (in *world space*)
- Light position (in *world space*)
- Light intensity
- Diffuse albedo

What needs to be computed (`out`)

- Diffuse fragment color

When dealing with vectors, we encourage you to use built-in GLSL functions. See this page for a complete list of GLSL math routines. Also note that there is *no* constant π in GLSL so you will have to `#define` it yourself.

2.4 Phong fragment shader (*25 pts*)

This task is effectively equivalent to Part 2.3, but substituting the functionality from your `PhongBSDF::eval()` routine in order to implement the Phong material. Create a new fragment shader in `src/shaders/phong.fs` and implement a shading routine for the normalized Phong BRDF model described in Part 1.2. If your offline Phong BSDF `eval()` function already works, this part is just a matter of translating your code to GLSL and applying the shading logic from Part 2.3 (and `SimpleIntegrator`):

Shader uniforms

- Camera position (in *world space*)
- Light position (in *world space*)
- Light intensity
- Diffuse albedo ρ_d
- Specular albedo ρ_s
- Phong exponent

What needs to be computed (`out`)

- Phong fragment color

Be wary of resources you find online regarding Phong: in our course, we implement the *Normalized Phong BRDF* model. The difference between this and the “original” Phong model (widely used in video games) is that the normalized version ensures **energy conservation**, which gives a more plausible result.

If you implemented both shaders correctly, you should get the exact same images as in the unshadowed offline renderer.

2.5 Bonus: Shadow mapping (*20 pts*)

Unlike with ray-tracing, computing accurate shadows can be very challenging in an interactive renderer. We do not have access to the scene BVH, let alone ray-tracing routines, in the shader. Various techniques address this problem, the most well-known of which is the *shadow mapping* algorithm.

The idea behind shadow mapping is simple: you will rasterizer your scene twice — you first rasterize the scene from the point of view of the light source, before performing the more traditional rasterization pass from the point of view of the camera. In the first rasterization pass, you need only store the rasterized depth buffer (*z*-buffer), which we call the **shadow map**. Everything that the light “sees” is lit, while objects occluded from the point of view of the light must lie in shadow. In the shader you use when rasterizing the scene from your camera viewpoint, you can transform points visible from the camera into the coordinate system of the light's shadow map, and then perform a depth comparison to determine whether the current pixel is in shadow or not.

This technique is quite involved — *only attempt this bonus if you have already completed the other questions*. You can read more on how to implement shadow mapping here. If you have a working implementation of shadow mapping, add an extra entry to the list of renders in your `config.json` submission file and submit it with your other renders. Choose only one of the two provided final scene.

```
{
  "scene": "Bonus: Teapot with shadow mapping",
  "render": "renders/teapot_bonus.exr"
}
```

When submitting, **make sure that the non-bonus version of your code compiles and is the default.** If you choose to do the bonus, use the `scene.config.bonus` boolean to toggle shadows. There is a separate `.toml` file `teapot_simple_realtime_bonus.toml` where `scene.config.bonus` is set to true.

What to submit

Render all the scenes in the `data/a2` directory. When complete, edit the `config.json` file with your credentials and submit all your code in a `.zip` or `.tar` archive file. Please create a *separate* directory with only the files you need to submit, and compress this directory — this is different than selecting all files to submit and right-clicking compress. Include your raw `.exr` files in separate folders (see the structure below).

```
a2_first_last.zip
   config.json
   src/
   offline/
      teapot_simple_offline.exr
      cbox_simple_offline.exr
   realtime/
      teapot_simple_realtime.exr
      cbox_simple_realtime.exr
      teapot_bonus.exr // Optional
```

Make sure your code compiles and runs for both online and offline parts before submitting! **Code that does not compile or does not follow this exact tree structure may result in a score of zero.** If your assignment does not compile at the moment of submitting but you do have some correct images from previous iterations, please submit these. You can use the `tree` command to verify this structure.