ECSE 446/546: Realistic/Advanced Image SynthesisAssignment 2: Point Lights & Shadows

Due Monday, October 1st, 2018 at 11:59pm EST on myCourses

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

Contents

In this assignment, you'll implement two basic reflectance models and a simple shading algorithm for these materials. To do so, you'll code a lighting “integrator” but, in the examples here, no actual integration is taking place; in the future, more complex light transport shading algorithms will actually solve integrals (numerically), and so the integrator nomenclature will become more appropriate. You'll also write a real-time shader for direct illumination, for the interactive part of this assignment.

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 the emitter we ask you to implement in this assigment is based on a simple, idealized model (as opposed to physically-realizable emitters), the shadows they will generate will be “hard”: they will not have any penumbra. In other words, every point in the scene is either completely lit or completely shadowed.

Start by copying your rendering loops from A1 to `src/core/renderer.cpp`. From now on, reference images provided for the assignments will be rendered using *multiple samples per pixel*. If you completed the A1 bonus, your rendered images should match the reference images exactly. Otherwise geometric/silhouette aliasing should be the only noticeable difference. You will *not* lose points for this.

1 Offline rendering (*50 pts*)

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 function `eval()` which currently returns black:

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

In TinyRender, the albedo ρ of a diffuse BRDF is casted as a constant 2D texture which allows for better integration with Bitmap textures. To evaluate the albedo at a surface interaction point `i`, you will have to invoke the texture's evaluation function by calling `albedo->eval(worldData, i)`. Implement `DiffuseBSDF::eval()` as follows:

1. Check that the incoming and outgoing rays are headed in the correct directions; if not return black.
2. Otherwise return the evaluated albedo divided by π, and multiplied by the cosine factor $\cos\theta_i$.

Use the `Frame::cosTheta()` function on `i.wo` and `i.wi` to perform your checks for the first part. These checks are necessary to avoid returning a nonzero color if a ray hits the backface of a triangle, or if the BRDF is being evaluated in a light direction under the surface's tangent plane. You are encouraged to draw a diagram with pen and paper to determine what needs to be checked.

All BRDF evaluation functions return the BRDF value times the cosine factor, that is, the BRDF term and the cosine term are bundled together. Furthermore, note that the implementation of the functions `sample()` and `pdf()` are 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 shading model that combines diffuse and glossy reflections. Many more advanced/accurate models exist (*e.g.*, using 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 system.

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

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

- ρ_d is the *diffuse reflectivity*, the fraction of the incoming energy that is reflected diffusely (clamped to 1 for energy balance),
- ρ_s is the *specular reflectivity*, the fraction of the incoming energy that is reflected specularly,
- n is the *Phong exponent* (higher values give more mirror-like specular reflection),
- α is the *specular angle* between the perfect specular reflect direction and the lighting direction (0 if negative).

Similar to the previous part, implement `Phong::eval()` in the file `bsdfs/phong.h`. The current implementation also returns black; your goal is to modify it as follows:

1. Check that the incoming ray is not hitting a backface (just like diffuse).
2. Evaluate the Phong BRDF using Equation (1).
3. Return this value multiplied by the cosine factor.

You can compute the perfect specular direction using the function `PhongBSDF::reflect()`, or from scratch. The file `core/platform.h` contains some definitions that you will need to use, such as π and $\frac{1}{2\pi}$. Note that `diffuseReflectance`, `specularReflectance` and `exponent` are also casted as textures, so you will need to evaluate these accordingly. This abstraction will allow us to use “shiny” textures, such as a varnished wooden floor.

1.3 Unshadowed direct illumination (*20 pts*)

To test your new reflectance models, you will implement two simple shading algorithms, one that computes unshadowed shading and another that computes images with (hard) shadows; in both cases, you will support shading from an isotropic point light source, one that emits the same radiance in all directions.

The integrator abstraction encapsulates shading algorithms: the complex shading algorithms we'll see later on the course will solve integral equations, such as the reflection equation. The shading algorithms you'll implement below, whose logic will be contained primarily in the `SimpleIntegrator::render()` routine of the `SimpleIntegrator` class, will compute a much simpler reflection equation that doesn't actually require any explicit integration: when we substitute the delta lighting models into the L_i term in the reflection equation, the shading equation simplifies to

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

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 direct illumination integrator 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 and evaluate the incident radiance to compute the outgoing radiance contribution. In other words:

1. Retrieve the light position $\mathbf{p}$ and intensity I.
2. Intersect your ray with the scene geometry.
3. If an intersection `i` is found, retrieve the hit surface material using `getBSDF(i)`.
4. Map the incoming direction `i.wi` to local coordinates by using the hit point `frameNs.toLocal()` transform.
5. Evaluate the BRDF locally and the light, and set the `Li` term accordingly.

Transforming your incoming direction is necessary to evaluate the BRDF, as it assumes the frame is in local coordinates (at the origin):

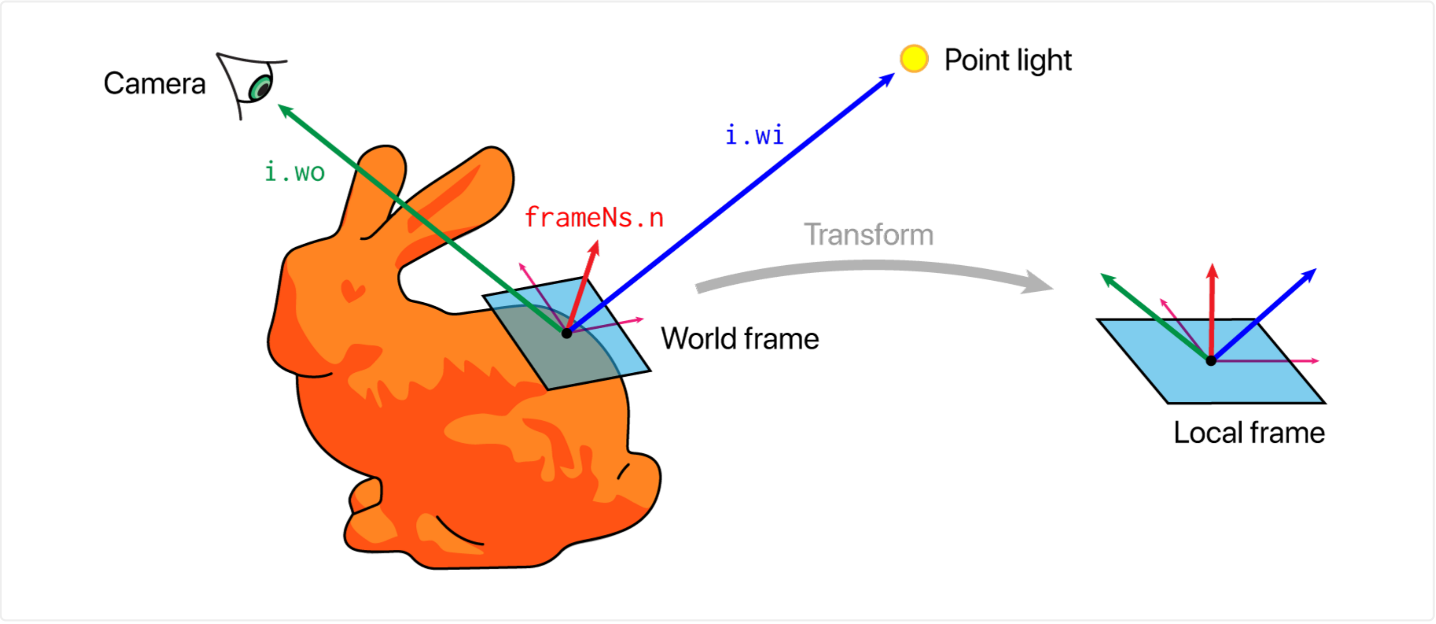

Figure 1: *Mapping a world frame to a local frame.*

The incident radiance L_i needs to be computed directly in the integrator. To avoid dedicating an entire structure to abstract emitters, TinyRender assumes that the light is "attached" to a very small quad mesh with 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 this radiant intensity to radiant flux by accounting for a distance falloff. This outgoing radiance depends on both the radiant intensity I and the squared distance between $\mathbf{x}$ and the light's position $\mathbf{p}$:

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

Since outgoing radiance from the light equals incident radiance at the shade point (*i.e.* $L_o(\mathbf{p}, -\omega_i) = L_i(\mathbf{x}, \omega_i)$), you have everything you need to implement the simple integrator. When you are done, render the scenes in `data/a1/sphere/tinyrender`—below are reference images for these two offline scenes:

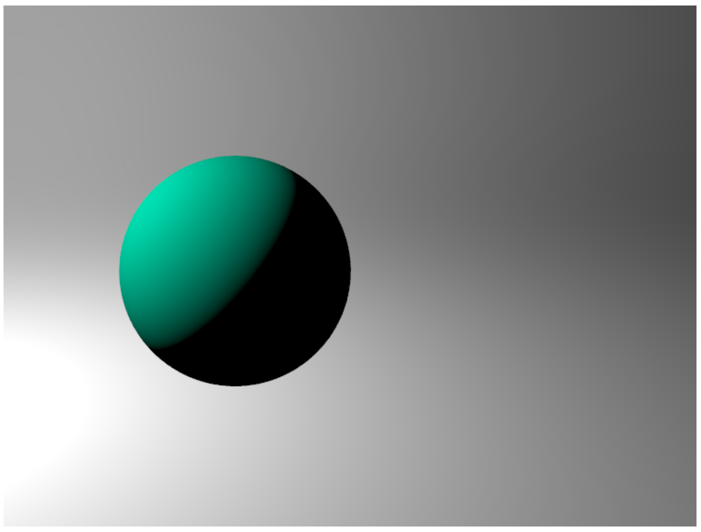

Figure 2: *Unshadowed diffuse sphere scene.*

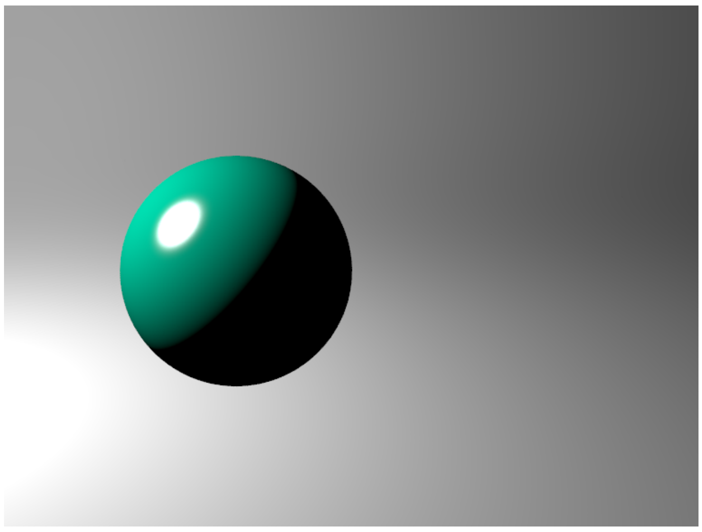

Figure 3: *Unshadowed Phong sphere scene.*

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

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

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

which can be implemented 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** intersection exists (rather than having to find the closest one). To evaluate visibility, create a shadow ray with origin at shading point $\mathbf{x}$, in the direction $\mathbf{p} - \mathbf{x}$ (normalized). Since you want to test for occlusion between $\mathbf{x}$ and $\mathbf{p}$ only, you need to set the ray's `max_t` value accordingly.

Make sure to compare your results with the reference images below.

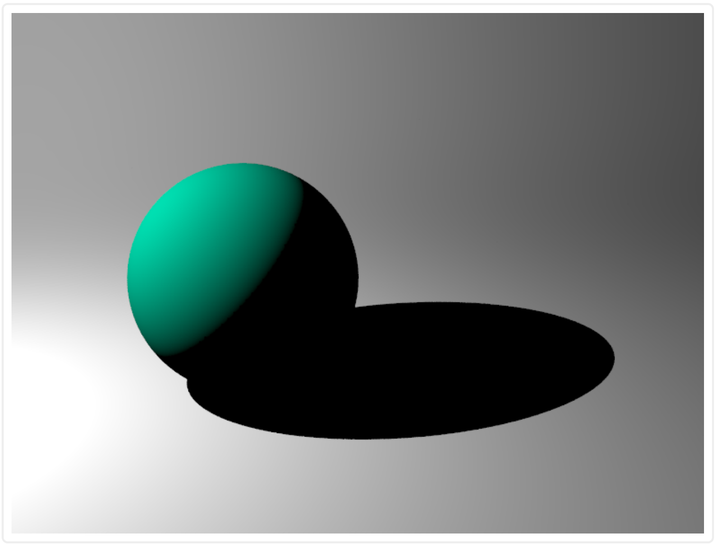
Figure 4: *Diffuse sphere scene with shadows.*

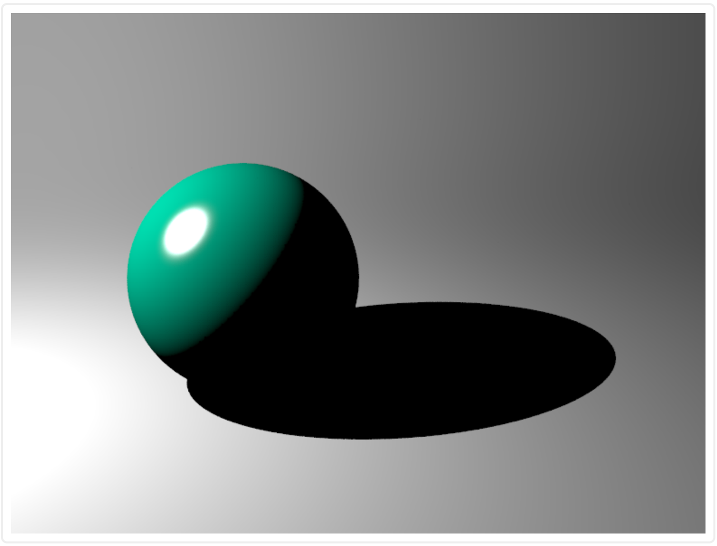
Figure 5: *Phong sphere scene with shadows.*

2 Real-time rendering (*50 pts*)

The real-time part of this assignment is very similar to the offline part, as both frameworks should yield the same images. Once again, your scene will have two types of materials (diffuse and Phong) and a single point light. Both materials require you to implement their corresponding fragment shaders; the point light illumination is implemented within these two shaders. A single vertex shader will be shared across both cases.

While it is possible (and more efficient) to do all your shading computations in *view space*, for this assignment do all your shading computations in *world space*.

2.1 Render pass (*5 pts*)

Similar to A1, complete the implementation of `src/renderpasses/direct.h` as follows:

1. Pass the light position and intensity via GL uniforms.
2. Pass shader-specific parameters via GL uniforms.
3. Draw the object using the shader corresponding to its material.

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, the appropriate `glUniform` should be used.

The variable `GLObject.shaderIdx` contains an index determining which shader is being assigned (diffuse or Phong):

```
DIFFUSE_SHADER_IDX
PHONG_SHADER_IDX
```

Check which shader you're dealing with and create specific `glUniform`'s in each case. When you're done, don't 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 called `shaders/simple.vs` to your TinyRender codebase. As mentioned earlier, this shader will bere used with every fragment shader. This vertex shader should be very similiar to your shader from A1:

Shader uniforms

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

Attributes (`in`)

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

What needs to be computed (`out`)

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

Implement this vertex shader and don't forget to set `gl_Position` with the screen space vertex position.

2.3 Diffuse fragment shader (*15 pts*)

Next, implement a fragment shader for the diffuse material in a new file `src/shaders/diffuse.fs`. At a high level, you will imlement the shading logic from `DiffuseBRDF::eval()` and your `SimpleIntegrator` light evaluation routine into this single, monolithic shader. This is where you get to use the `glUniform`'s you previously created in Part 2.1.

Shader uniforms

- Camera position (in *world space*)
- Light position (in *world space*)
- Light intensity
- Albedo

What needs to be computed (`out`)

- Diffuse fragment color

When dealing with vectors, resorting to built-in GLSL functions is allowed and encouraged. See this page for a complete list of mathematical functions allowed. 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 part is more or less the same as Part 2.3, but using your `PhongBSDF`. Create the fragment shader `src/shaders/phong.fs` and implement 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 reflectance ρ_d
- Specular reflectance ρ_s
- Phong exponent

What needs to be computed (`out`)

- Phong fragment color

Be wary of resources you find online regarding Phong: in this course, we implement the *Normalized Phong BRDF* model. The difference between this and the original Phong (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 image as with the offline renderer (without shadows).

2.5 Bonus: Shadow mapping (*40 pts*)

Shadows are hard to do in real-time rendering since tracing shadow rays is not an option; we don't even have access to the BVH of the scene in the shader.

Various techniques have been invented in the past to approximate the appearance of shadows in real-time. The most well-known technique is called *shadow mapping*.

The idea of the technique is simple: you render the scene from the point of view of the light source and store the resulting image in a depth buffer that we call the **shadow map**. Everything that the light "sees" is lit, while what it doesn't see must lie in shadow. In a shader, when you render the scene from the point of view of the camera (like what you are used to), you use this buffer to check whether the current pixel is in shadow or not.

This technique is quite involved. As such, *only attempt this bonus if you've already completed the other questions and you are confident in your C++/GL skills*. 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 your list of renders in your submission `config.json` 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"
}
```

What to submit

Render all the scenes in the `data` directory and create your HTML submission using the assignment submission script. When you're done, submit your slider file `a2_id.html` and all your code in a `.zip` or `.tar` archive file. Include your raw `.exr` files in separated folders (see structure below).

```
a2_first_last.zip
   a2_id.html
   src
   offline/
      teapot_simple_offline.exr
      cbox_simple_offline.exr
   realtime/
      teapot_simple_realtime.exr
      cbox_simple_realtime.exr
      cbox_or_teapot_bonus.exr // Optional
```

Make sure your code compiles and runs for both online and offline parts before submitting! You will obtain a score of zero if your submission does not follow this exact structure. You can use the `tree` command to verify this.

formatted by Markdeep 1.04