CS148 Homework 3

Homework Due: Jul 16th at 11:59 PM PST
Quiz Date: Tuesday Jul 12th

1 Assignment Outline

This assignment is split up into:

1. a coding part that takes place all in the same Python script and has you finish the code for a simple ray tracer. The parts of the code that you will implement for this simple ray tracer are:
   - `TODO SPHERE-RAY INTERSECTION`: Compute the intersection of a ray and an implicit sphere surface as the value of a parameter t that takes our ray from the camera, through a pixel, and to the sphere object.
   - `TODO INTERSECTION POINT`: After computing the sphere-ray intersection t value, use it to compute the actual 3D intersection point between the ray and the sphere object.
   - `TODO DIFFUSE BRDF`: Compute the diffuse color component of the outgoing radiance from the intersection point based on the interaction between the light and object surface.
   - `TODO SHADOW RAYS`: Cast a shadow ray from the sphere-ray intersection point to each of the lights to determine whether any of the lights are occluded, and if so, ignore them for the lighting computations.
2. a Blender part where you will be experimenting with all 4 types of light that Blender offers:
   - `TODO POINT LIGHT`
   - `TODO AREA LIGHT`
   - `TODO SPOTLIGHT`
   - `TODO DIRECTIONAL LIGHT`
   - `TODO ALL LIGHT`: Create a scene with your own object that includes all 4 different types of light: point, area, spot, and directional lights.

These TODOs are marked with the `Action:` indicator in this PDF as well as TODO in the section headers.

Please download `HW3.zip`, save it in some course directory `$CS148_DIR` on your machine, and unzip the file in this directory. All the work will be done in the `$CS148_DIR/HW3/HW3.ipynb` Jupyter Notebook. When you are finished, print and save your Notebook as a PDF and submit to Gradescope.

2 Quiz Questions

- How does the distance between a light and an object as well as the tilt angle between the two affect the irradiance on the surface of the object? How does the concept of a tilt angle also come up for the radiance of a light?

- What causes color bleeding in real life? Give a high level description of what we would need to do with our objects in a scene to model color bleeding.
- Describe one of the three ways we discussed in class on how to compute a ray-triangle intersection. You don't need to explain all the mathematical details, but give enough details to explain the high level idea of each step.
- What is the idea of bounding volume hierarchies, and why do they help speed up ray tracing? When might we prefer uniform partitions instead and why?
- Why is transforming a normal vector between object and world space different from transforming an object? What is one reason why we want the normal vector in world space coordinates?

3 (Simple) Ray Tracer

The topic of simple, non-recursive ray tracing was covered in Lecture 6. All the TODOs for the coding part of this assignment should be doable from just the lecture material.

Open up the Jupyter Notebook in `$CS148_DIR/HW3/HW3.ipynb`, run the first cell as usual to load in our libraries, and then take a look at the second cell. This cell contains a full implementation of a simple, non-recursive ray tracer with some parts missing. You will be writing code to fill in the missing parts.

We recommend taking some time to look over the code first to get a sense of how it's setting up the ray tracer. Use the code comments as guides for understanding the code. As always, if you have any questions, feel free to ask on Piazza. Questions related to code that we provide can be asked publicly.

Note that if you run the cell as is, then after some processing, the cell should output a black image both in the notebook and in `$CS148_DIR/HW3/images/output.png`.

3.1 Your Task (TODO SPHERE-RAY INTERSECTION)

Action:

This should be the first part that you do. Look for `TODO SPHERE-RAY INTERSECTION` in the cell and read through the code comments on what to do. You need to complete the function for computing a sphere-ray intersection, given the camera aperture, ray direction, sphere center point, and sphere radius as input. The output is the t value that takes our ray from the camera aperture to the intersection point on the sphere. Use the quadratic approach as discussed in lecture.

3.2 Your Task (TODO INTERSECTION POINT)

Action:

After completing the sphere-ray intersection, look for `TODO INTERSECTION POINT` in the cell. You will now need to use the t value returned from the sphere-ray intersection function to compute the actual 3D coordinates of the intersection point. Think back to our ray equation, and don't be afraid to use any variables that we already defined for you. This part can be as simple as 1 line of code!

After completing this TODO, you should expect some change in the output if you run the cell. The cell saves the image output to `$CS148_DIR/HW3/images/output.png`.

A correct implementation will output an image similar to that of Figure 1. You can change the screen width and height variables to increase the resolution of the output image if you want (though it will make the code take longer to run).

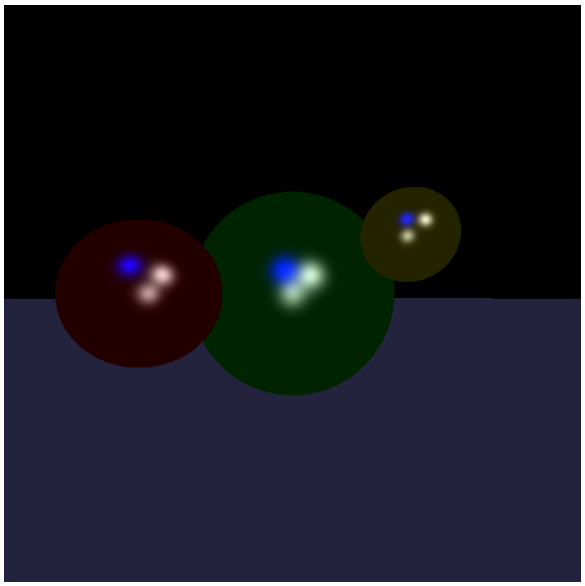


Figure 1: The image result after computing the sphere-ray intersection point in the code rendered at 640x640 resolution.

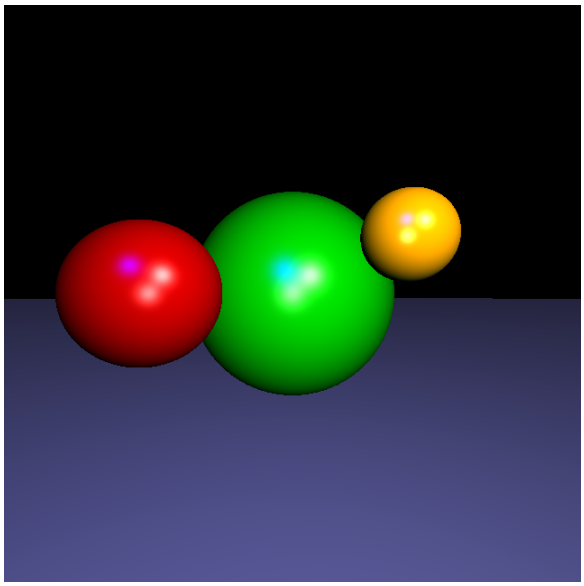


Figure 2: The image result after adding diffuse lighting to the result on the left rendered at 640x640 resolution.

3.3 Your Task (TODO DIFFUSE BRDF)

Action:

Look for `TODO DIFFUSE BRDF` in the cell. Follow the code comments for directions on how to compute the diffuse lighting for the intersection point based on the Blinn-Phong BRDF. This part can also be as simple as 1 line of code.

A correct implementation will output an image similar to that of Figure 2.

3.4 Your Task (TODO SHADOW RAYS)

Action:

Finally, look for `TODO SHADOW RAYS` in the cell. Again, use the code comments for guidance. A skeleton of the loop that you need to write has also been provided.

This part has you mimic the code in between the `START REFERENCE` and `END REFERENCE` comments to cast a shadow ray from the intersection point to the light. If the shadow ray intersects an object between the light and intersection point, then the radiance from that light is occluded. In that case, the code should just move onto the next light, thus skipping any lighting computations with that light.

After this TODO, you should get the final, ray traced image with shadows, similar to Figure 3.

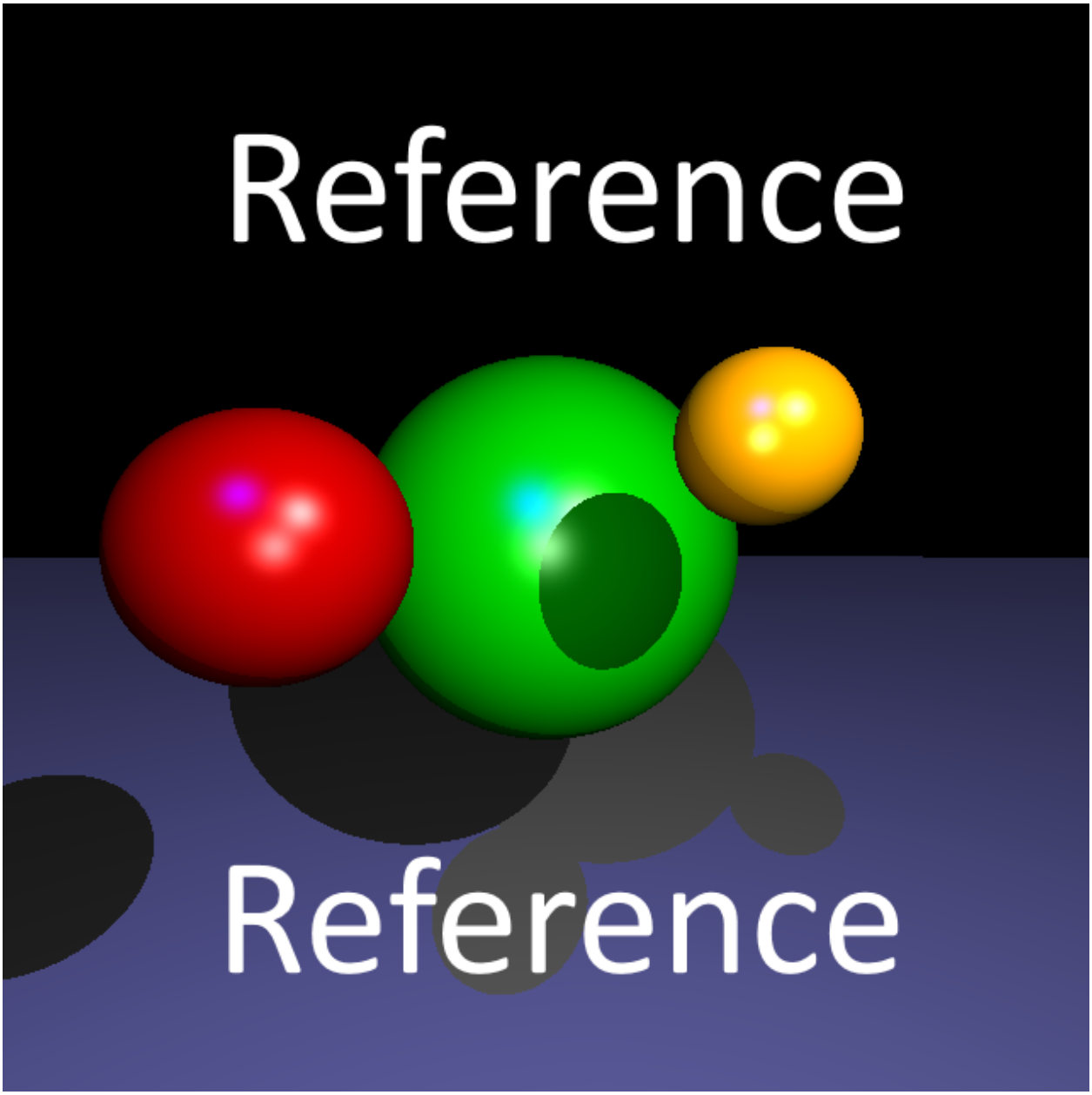


Figure 3: The final ray traced image after adding shadows to the result from Figure 2 rendered at 640x640 resolution.

4 Lighting in Blender

Blender by default provides 4 different types of lights for you to place into your scene: point, area, spot, and directional lights. We covered 2 of these in Lecture 5 (point and area), and the other 2 are pretty intuitive to pick up.

There are two ways to add any of these lights to your scene. One option is to add a new **Light** from the **Add** menu. The other is to modify an existing light in your scene in the Properties Editor under **Object Data Properties**. There, you can change your light between all 4 types.

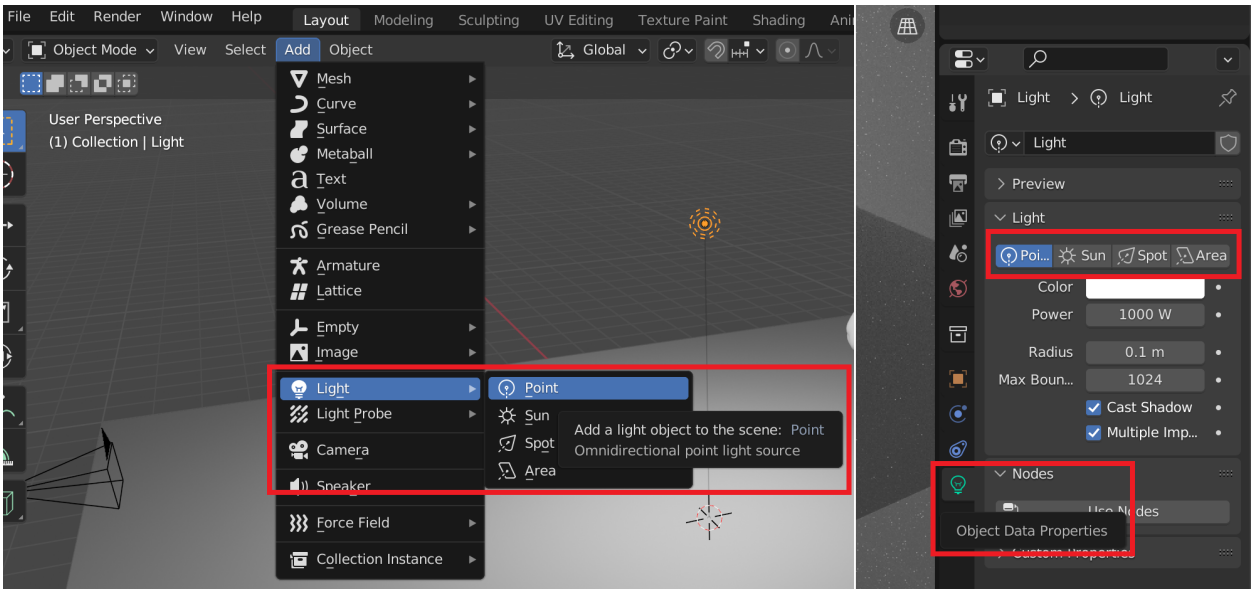


Figure 4: Ways to add a particular light type to your scene.

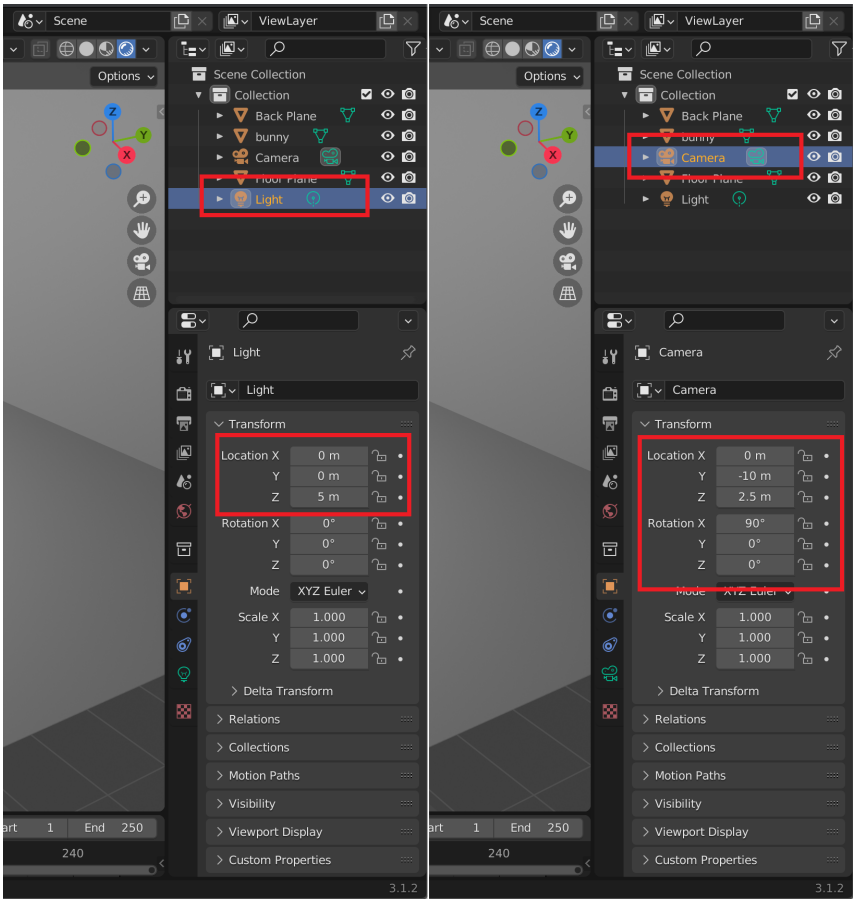


Figure 5: Transforming a light or camera using the Properties Editor. Note that in the case of a point light, only translations matter. Scaling has no effect on either.

4.1 Your Task (TODO POINT LIGHT)

Action:

First, before we experiment with the 4 light types, let's set up an actual scene. We'll use a point light for our lighting in this set up, since it's the simplest to understand.

Load in your own .obj file(s) and arrange the object(s) (i.e. translate, rotate, scale them) in the Blender Viewport however you want for a coherent scene.

Then, make sure to add a point light (can use the default), and position it where you see fit. Note that you can transform a point light just like you can an object (e.g. with the Properties Editor, hot keys, etc). Figure 5 shows how to do so with the Properties Editor. The same can be done with the camera.

You may want to toggle the camera view back and forth when arranging your object(s), light(s), and camera as shown in Figure 6. Figure 6 also shows an example of a simple scene if you need ideas on setting yours up. This scene just consists of the Stanford bunny on top of a plane in front of another plane. The camera is facing the bunny straight down the Blender y-axis, and the point light is placed above the bunny.

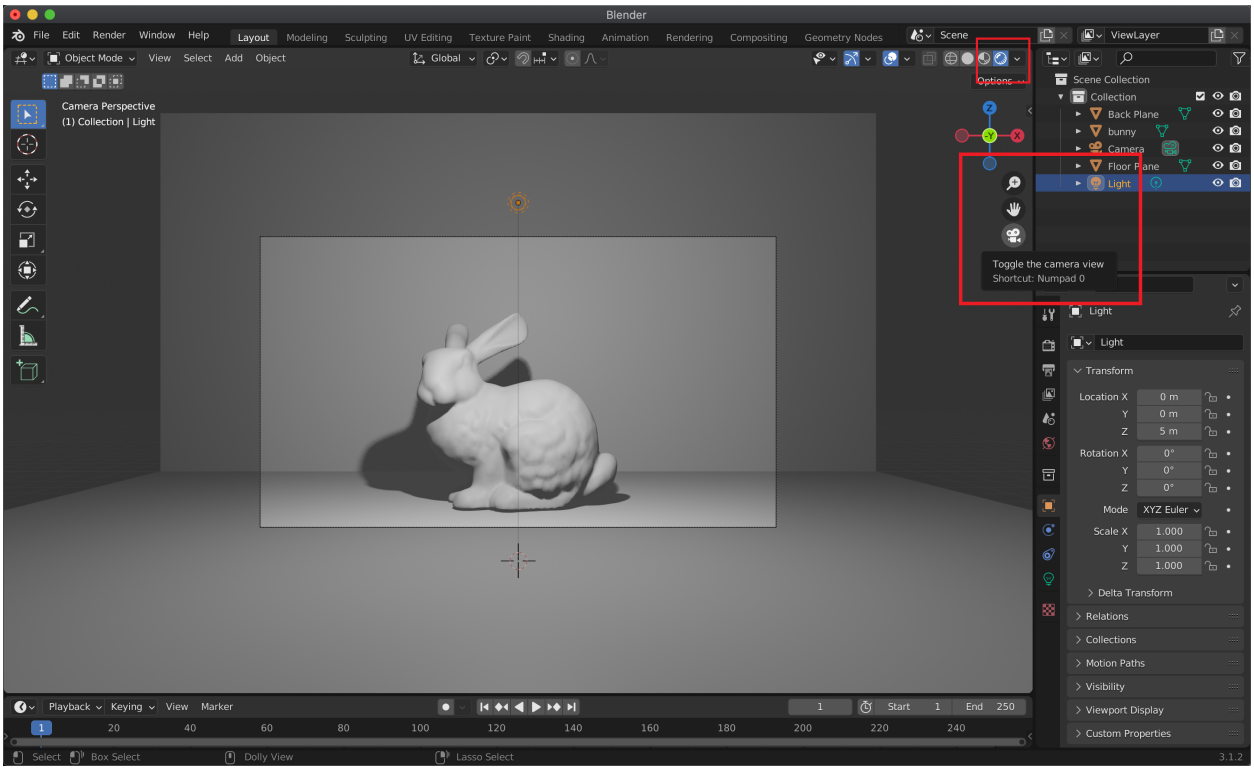


Figure 6: Toggling to camera view in the Blender Viewport. The button for toggling the render preview is also located in the Viewport toolbar right above the camera view toggle.

When you're satisfied with your scene setup, go to **Render Properties** in the Properties Editor and change the **Render Engine** to **Cycles**. **This tells Blender to use its ray tracing engine**, as opposed to **Eevee**, which is its scanline renderer. You can preview the render with the right-most button in the Viewport toolbar above the camera view toggle. Note though that Blender may start running very slowly if you try to make scene edits while previewing the Cycles (ray traced) render! As mentioned in lecture, ray tracing isn't suited for real time rendering.

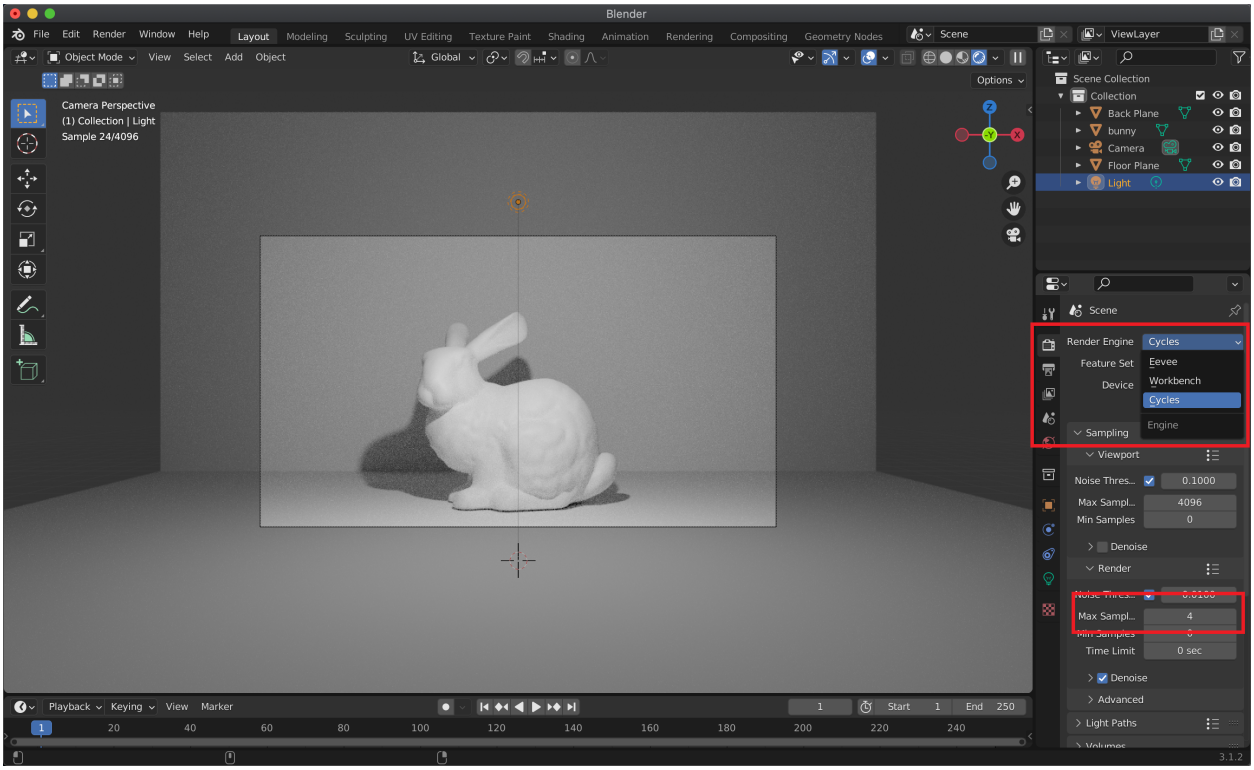
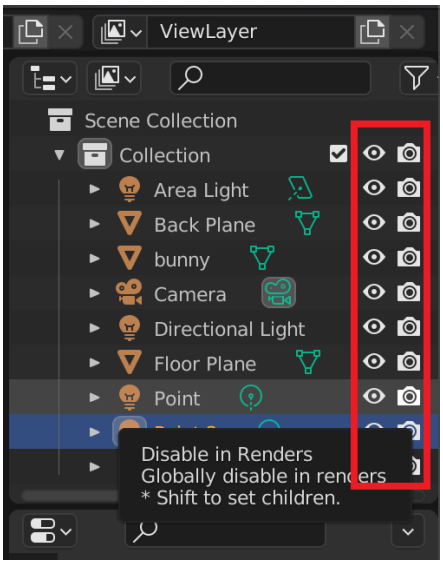


Figure 7: **Cycles** is Blender's ray tracer, as opposed to **Eevee**, which is its scanline renderer. By default, Blender has the samples for Cycles set to some large number. You want to change it to a much lower number, e.g. 4, to make the render faster.

After setting Blender's render engine to **Cycles**, make sure to change the default **Max Samples** from 4096 to a much lower number like 4. We'll talk about sampling later in the class, but for now, know that the max samples is basically how many times Blender will repeat the render for better results. Too high of a number will cause your render to take an incredibly long time.

Finally, render your scene as an image. You can go to **Render** near **File**, then **Render Image**, or press the **F12** hotkey. When the Blender Render window finishes, save the result under **Image** to the **$CS148_DIR/HW3/images** directory. Find the **TODO POINT LIGHT** cell in the Jupyter notebook (**$CS148_DIR/HW3/HW3.ipynb**), and edit the cell to display your saved image.



4.2 Your Task (TODO AREA LIGHT)

Action:

Disable your point light from Section 4.1 (using the eyeball and camera icons in the **Scene Collection** tree above the Properties Editor), and add an area light instead. Transform the area light however you see fit for your scene, and render another image with just the area light as your source of light. Find the **TODO AREA LIGHT** cell, and edit the cell to display your saved image.

Notice how the area light results in a much less uniform spread of light, as all the light rays are restricted to coming from a particular area. This is in contrast to the point light, where light rays are sent out indiscriminately in all directions around the point. However, the larger area of an area light results in softer looking shadows (e.g. less dark contrast) than what might result from a point light (see Figure 8 vs. Figure 9).

Figure 8: An example of a scene illuminated with only an area light. The light rays are restricted to coming only from a particular area, though having multiple rays going in the same direction can lead to softer shadows than what would be produced by a point light.

Figure 9: An example of a scene illuminated with only a point light. Point lights tend to produce much harsher shadows with darker contrasts to that of the surroundings, as each direction only gets one ray of light.

4.3 Your Task (TODO SPOTLIGHT)

Action:

Repeat Sections 4.1 and 4.2, but this time with a spotlight. Disable your point light and area light before proceeding. When you're satisfied, find the `TODO SPOTLIGHT` cell, and edit the cell to display your saved image.

Spotlights model lights that are engineered to emit light rays in a cone-like shape. Imagine lamps or streetlights in real life where the light is often surrounded by some sort of lampshade or outer object designed to concentrate the light. This leads to a light type that also aims its light rays at a specific area, like area lights, but still spreads its light rays in multiple directions, similar to point lights.

Spotlights are popular in modeling due to how common they tend to be in our everyday lives. In addition, they also are able to produce more dramatic lighting compared to a point light or area light. For instance, consider Figure 10 compared to Figure 9. It's much easier to concentrate the light from a spotlight on a particular part of a scene than a point light, whose light rays go in every direction.

Figure 10: An example of a scene illuminated with only a spotlight for more dramatic lighting. The spotlight excels at concentrating light all into one spot in the scene.

4.4 Your Task (TODO DIRECTIONAL LIGHT)

Action:

The last type of light that we'll examine is the directional light or "sunlight" as Blender calls it. The idea of directional lights is that they don't have their own locations (i.e. no actual positions in world space), but rather, they shoot light rays uniformly across all of space in the same direction. Think of the sun in real life. The sun is so far away from us that all its light rays at any given instant might as well be shining in the same direction. Hence why Blender calls them sunlights.

Mathematically, when we consider directional lights for lighting computations (i.e. the lighting equation), we just ignore any concept of distance and use the same direction for the light vector regardless of where we are in the scene. So unlike what happens with point lights or area lights, an object that's twice as far away as another from a directional light still gets the same amount of

light. Only the direction matters for a directional light.

Disable all your other lights and illuminate your scene with just a directional light. Notice how modifying its translation values makes no difference. Only changing its rotation matters, as that affects its direction. When you're satisfied, find the `TODO DIRECTIONAL LIGHT` cell, and edit the cell to display your saved image.

Figure 11: An example of a scene illuminated with only a directional aka "sun" light. Directional lights have no concept of position. Instead, they shine uniformly across all of space in one direction.

4.5 Your Task (TODO ALL LIGHTS)

Now try lighting up your scene with all 4 types of light. Try to get a feel of how each light might come up for what you want to do in your final project. From Figure 12, we see that using a mix of lights can lead to a more natural look with soft shadows and more even illumination. When you're satisfied, find the `TODO ALL LIGHTS` cell, and edit the cell to display your saved image.

Figure 12: An example of a scene illuminated with all 4 types of light: point, area, spot, and directional lights. We end up with a more natural look with soft shadows and more even illumination than from just using one light type.