

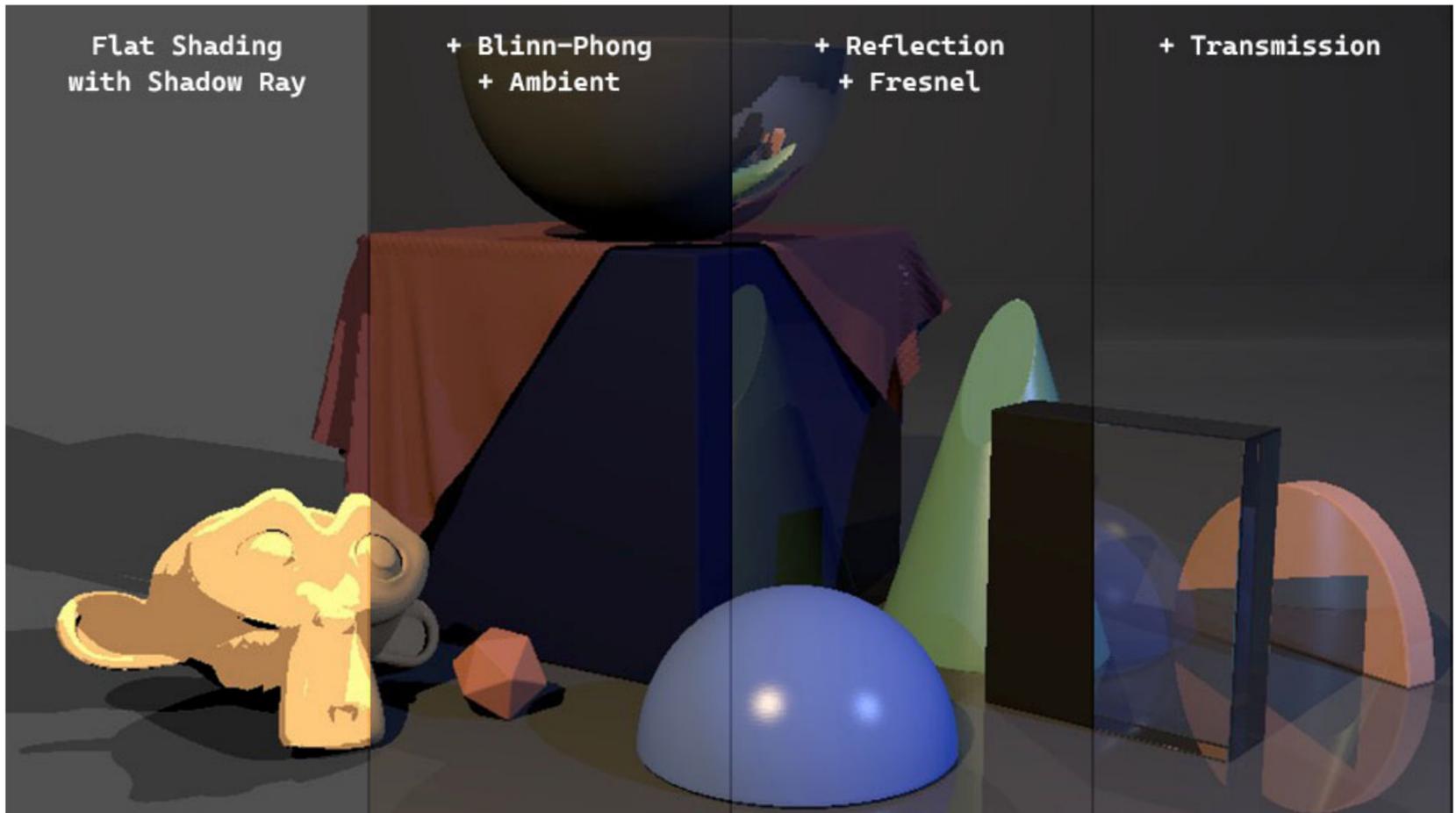
HW3 Ray Tracing

CS 148 Fall 2020-2021

Due Date: Monday, 5 October 2020 by 7pm PT

Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to CAs on Piazza or attend office hours on Nooks.

Be aware of the **Checkpoints** below. Make sure you complete each one since we will do the grading based on them.



Steps to the final image of this homework.

This week, we will create a ray tracer using Python in Blender. The scene is created in Blender, and we will use scripts to get information from the scene through Blender's Python API, and create our own routine for bouncing lights in the scene to get rendered images. The image above shows the four main steps to our final image. The power of ray tracing mostly lies in step 3 and 4, where we add reflection and transmission to the objects. In contrast, scanline renderers cannot handle them with physical correctness.

For shading, we will adopt a popular shading model called Blinn-Phong. It is a simple but sufficient (at least for this simple ray tracer) approximation of how light interacts with objects, and is adopted by OpenGL and Direct3D as the default shading model. Read more [here](#).

⚠ Start early! This assignment is more coding intensive than the others. Checkpoint 1-3 only require Tuesday's lecture, checkpoint 4-6 will need Thursday's lecture.

⚠ Please **read through this handout before you start coding**. We provide many helpful tips here.

⚠ There will be a lot of math and logic that you need to comprehend before writing your own ray tracer. If you get overwhelmed at any point, we suggest you review the lecture slides and make sure you understand the high-level idea.

First, download the .blend file for this homework.

http://web.stanford.edu/class/cs148/assignments/hw3_ray_tracing.blend

There's nothing more to it other than two text files, an already set-up scene, and a better workspace layout for doing scripting work. You can copy the scripts to a blank Blender file, add/delete objects and lights, and everything should work as well. However, please do not change anything in the scene for submission.

I. Scripting in Blender

In this homework, we will use Blender's scripting functionality to write a ray tracer.

I.1. Start Blender from command line



For Python development in Blender, it is easier to use the console/terminal since the Blender output will be there, making it easier for us to print debug information, such as `print()`. To use the console, we need to launch Blender from the command line. See the [Blender manual](#) for more information.

First, start Blender from the command line. The command line will be the place we see debug information.

▼ Windows

Press Windows+R to open the "Run" box. Type "cmd" and hit enter to open the command prompt.

First, `cd` to the Blender installation directory. The default location is `C:\Program Files\Blender Foundation\Blender 2.90`.

Then type `Blender.exe` to launch Blender.

```
C:\WINDOWS\system32\cmd.exe - blender.exe
Microsoft Windows [Version 10.0.19042.487]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\>cd "C:\Program Files\Blender Foundation\Blender 2.90"

C:\Program Files\Blender Foundation\Blender 2.90>blender.exe
Read prefs: C:\Users\\AppData\Roaming\Blender Foundation\Blender\2.90\config\userpref.blend
found bundled python: C:\Program Files\Blender Foundation\Blender 2.90\2.90\python
```

▼ Mac

Please refer to the official document :

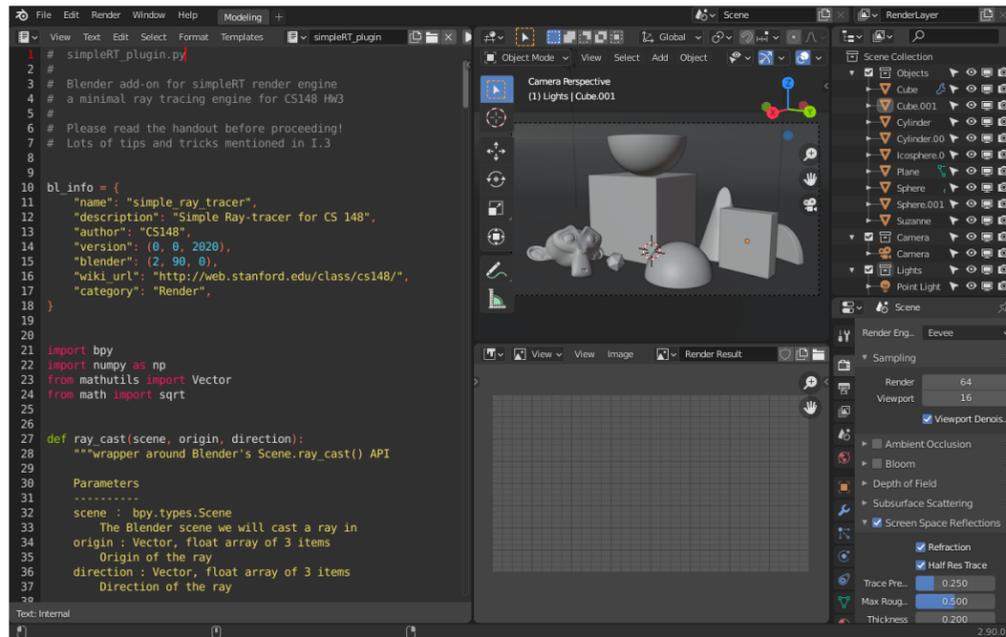
https://docs.blender.org/manual/en/2.90/advanced/command_line/launch/macos.html

▼ Linux

Please refer to the official document :

https://docs.blender.org/manual/en/2.90/advanced/command_line/launch/linux.html

Blender will launch as usual. Open the downloaded .blend file from the menu bar: `File→Open`. You should see the workspace layout like this:



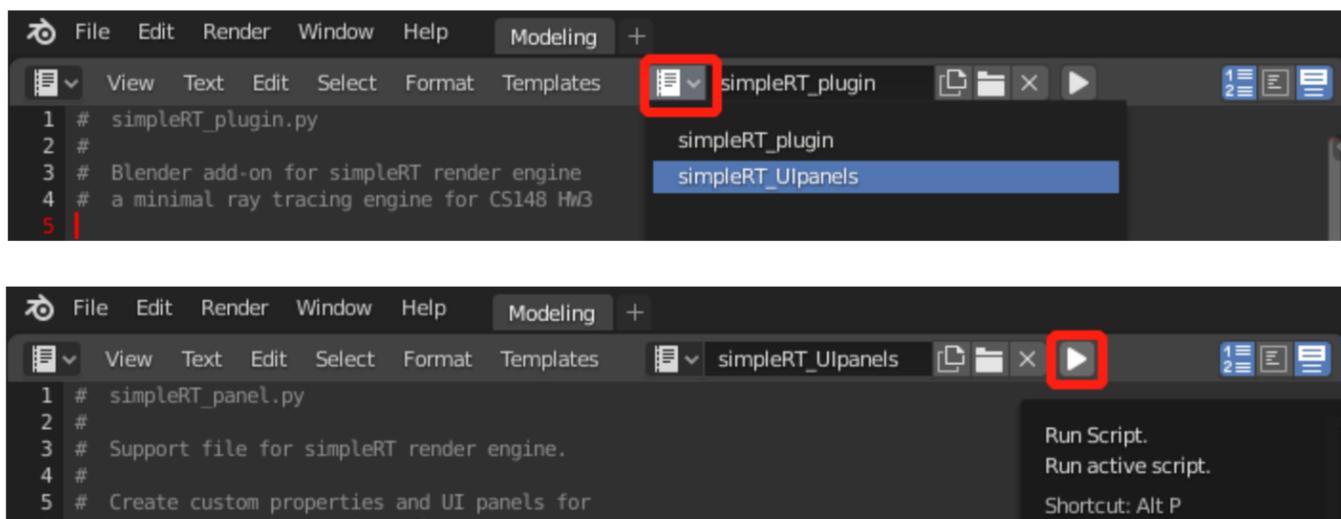
There are three main sections:

1. Left: Text Editor. This is where we edit and run the code.
2. Top right: 3D viewport. We can manipulate and preview the scene in this area.
3. Bottom right: Image Editor. We'll view our rendering results here.

1.2. The Text Editor

First, let's browse to a script called `simpleRT_UIpanels` and run it. In the Text Editor, you can switch between files using the dropdown button (fig 1). To run a script, press the "Run Script" button on the top bar of the Text Editor (fig 2). Alternatively, if you do not see the button, then you can still run the script from the menu bar above the text editor with "Text→Run Script". Nothing should appear in the command line if the script executed successfully.

 The `simpleRT_UIpanels` script creates custom properties that will be used for our renderer such as material properties, light parameters, etc.. It then exposes these properties through the Blender UI so we can change them easily from user interface widgets.

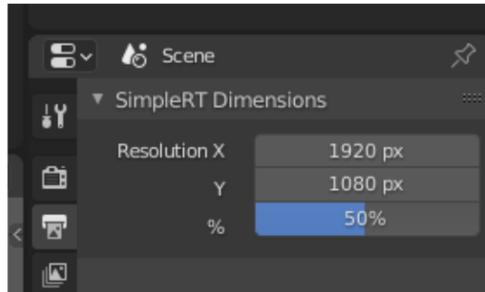
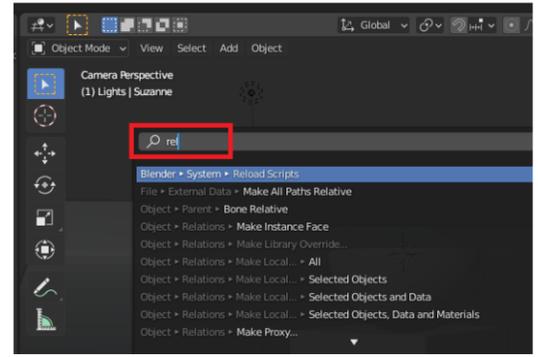
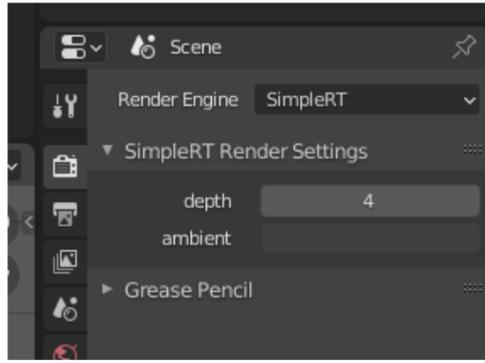
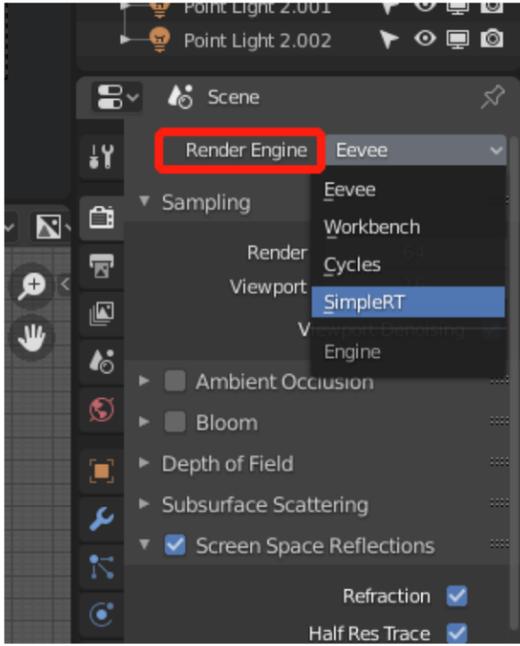


Then, switch back to the `simpleRT_plugin` and run it.

To check if everything is working:

From the Properties Editor, switch the Render Engine to `SimpleRT` (fig 1). This is the new render engine that we just added to Blender by running the script. We will work on this render engine in this homework.

The UI in Properties Editor should be refreshed (fig 2). You should be able to see the "SimpleRT Render Settings" panel in the Render tab, and the "SimpleRT Dimensions" panel in the Output tab. If you still see the UI panels from other render engines, press `F3` with your cursor in 3D Viewport, and type `reload script` and `Enter` to reload all the plugins (fig 3).

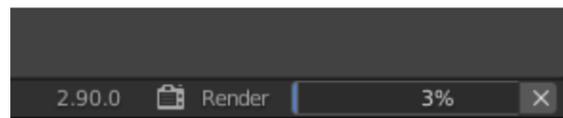


1.3. Start Coding

Now your task is to edit the `simpleRT_plugin` to finish all the TODOs.

After you make edits, run the script again, and hit **F12** to start the render. You should see the rendered image appear in the Image Editor and a progress bar in the bottom status bar (fig below). When rendering is done, use **Image → Save Image (Alt/Option S)** to save the result.

You can press **Esc** at any time to stop the render.



With the starter code, the image should render out completely black.

1.3.1. Editing and debugging

Blender has a very friendly development environment compared with other 3D software but it is still not on par with professional Python IDEs or professional text editors.

There are some handy shortcuts in the text editor. Formatting might be the most common operation: **Tab** for indent, **Shift+Tab** for unindent, and **Ctrl /** or **Cmd /** (mac) to toggle comments. For more shortcuts in Text Editor, please see [Blender doc](#).

Most debugging information will be displayed in the command line window you start Blender from, but Blender still displays some information from the user interface. There is a **Python Console** ([Blender doc](#)) that you can use as an interactive Python shell and the **Info Editor** ([Blender doc](#)) which displays logs, warnings, and error messages.

1.3.2. Speed up the render with lower resolution

To speed things up, we should keep the image resolution low so the renders are fast, which means faster iterations. After the low-resolution image looks correct, crank up the resolution and render a higher resolution image for submission.

A quick and easy way to change the resolution is to change the resolution percentage from **Properties Editor → Output Tab → SimpleRT Dimensions → %**. This will effectively take the original resolution and shrink it down. So you will render a smaller image, with the same aspect ratio.

Usually **25-50%** will be a good place to start, which will keep your render time under 10 seconds. This will of course give you a very aliased result, but you should be able to tell if your code is working from this image.

After the image looks right to you, change to **100%** for a higher resolution image. This will take around a minute. With this resolution, you will be able to check the details.

! For submission, use **100%** for resolution percentage, and keep resolution X and Y to **480** and **270** respectively (the default value for this .blend file).

I.3.3. Vector manipulation

In the code, we mostly use the data types that come with Blender. `mathutils.Vector` is the one we use to represent location, direction, and color. See basic usage [here](#).

The most common functions are:

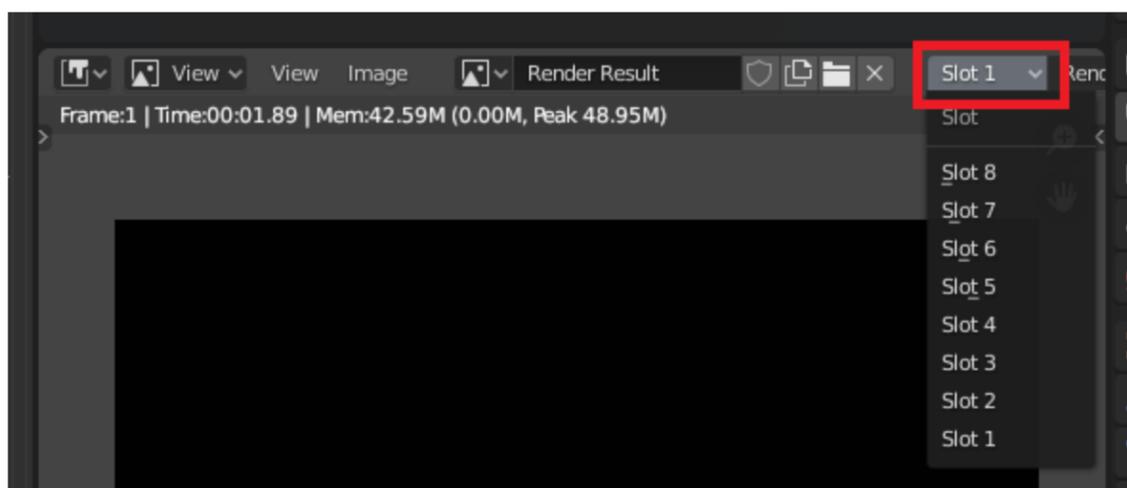
- `vec.normalized()` to return a normalized copy of `vec`
- `vec1.dot(vec2)` to take the dot product of two vectors
- `vec.length_squared` to get the squared vector length

To perform element-wise vector multiplication, the best way would be to convert the vectors to numpy arrays using `arr = np.array(vec)`, then use `arr1 * arr2` to get the result.

I.3.4. Compare rendering results

In the Image Editor, you can store up to 8 rendered images in slots and toggle between them afterward. This could be helpful when comparing the results from different versions of code, or different material properties.

To render image to a certain slot, select that slot before rendering:



To toggle between the slots, use the number keys to go to the slot with the corresponding number (remember to turn off `Emulate Numpad` from the Preferences!), or press `J` and `Alt J` to cycle forwards and backward through saved renders, respectively. See [Blender doc](#) for more.

II. Checkpoints

All the TODOs are listed as comments in the Python scripts. Please read them to get started. Here we will provide general guidelines and checkpoint images you can compare with.

The ray-object intersection will be taken care of by the `scene.ray_cast` function from Blender. Our focus will be on how to use the intersection point and surface normal to figure out color calculations and light bouncing.

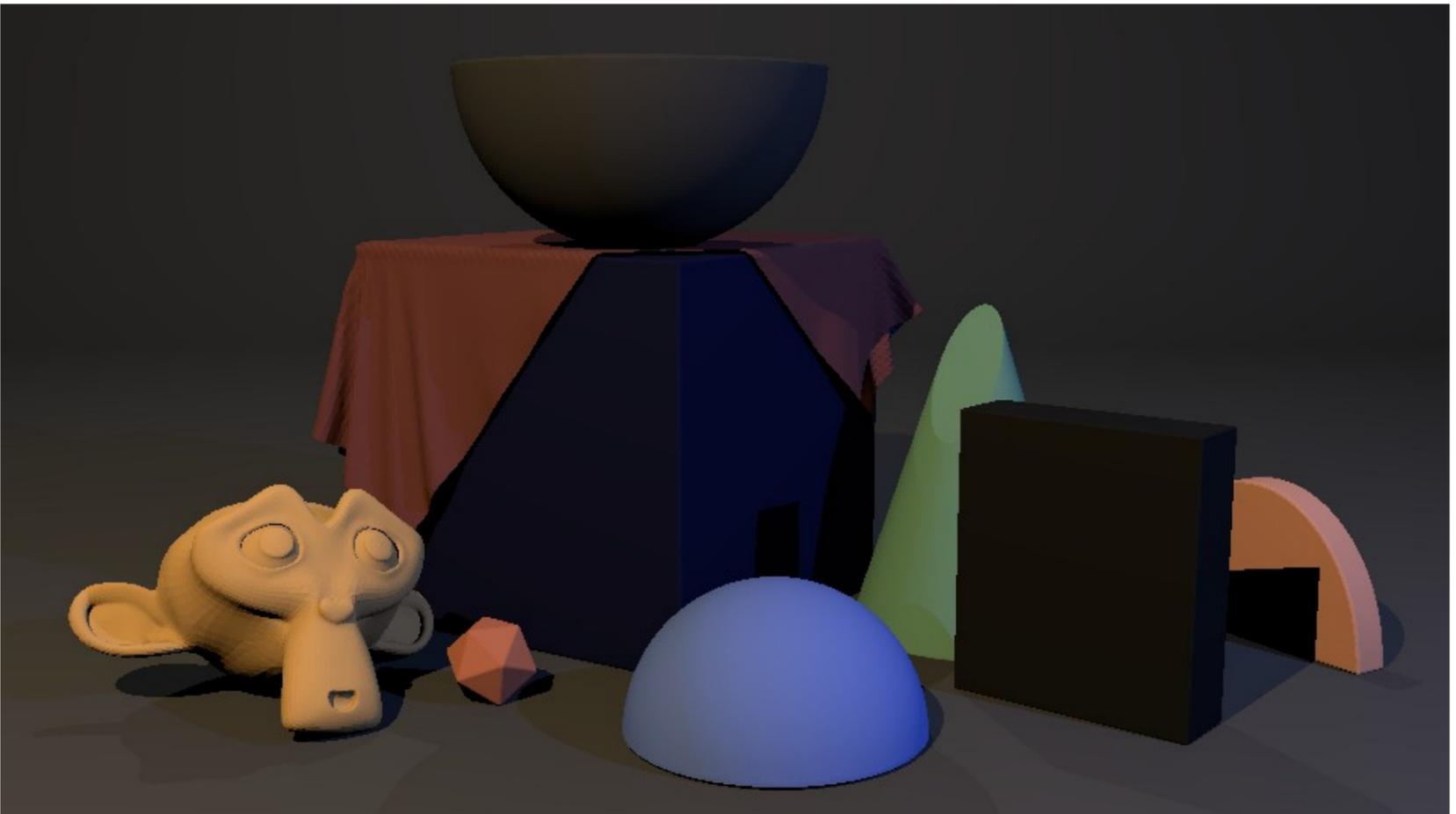
II.1. Shadow Ray



Checkpoint 1: Save the image with shadow ray.

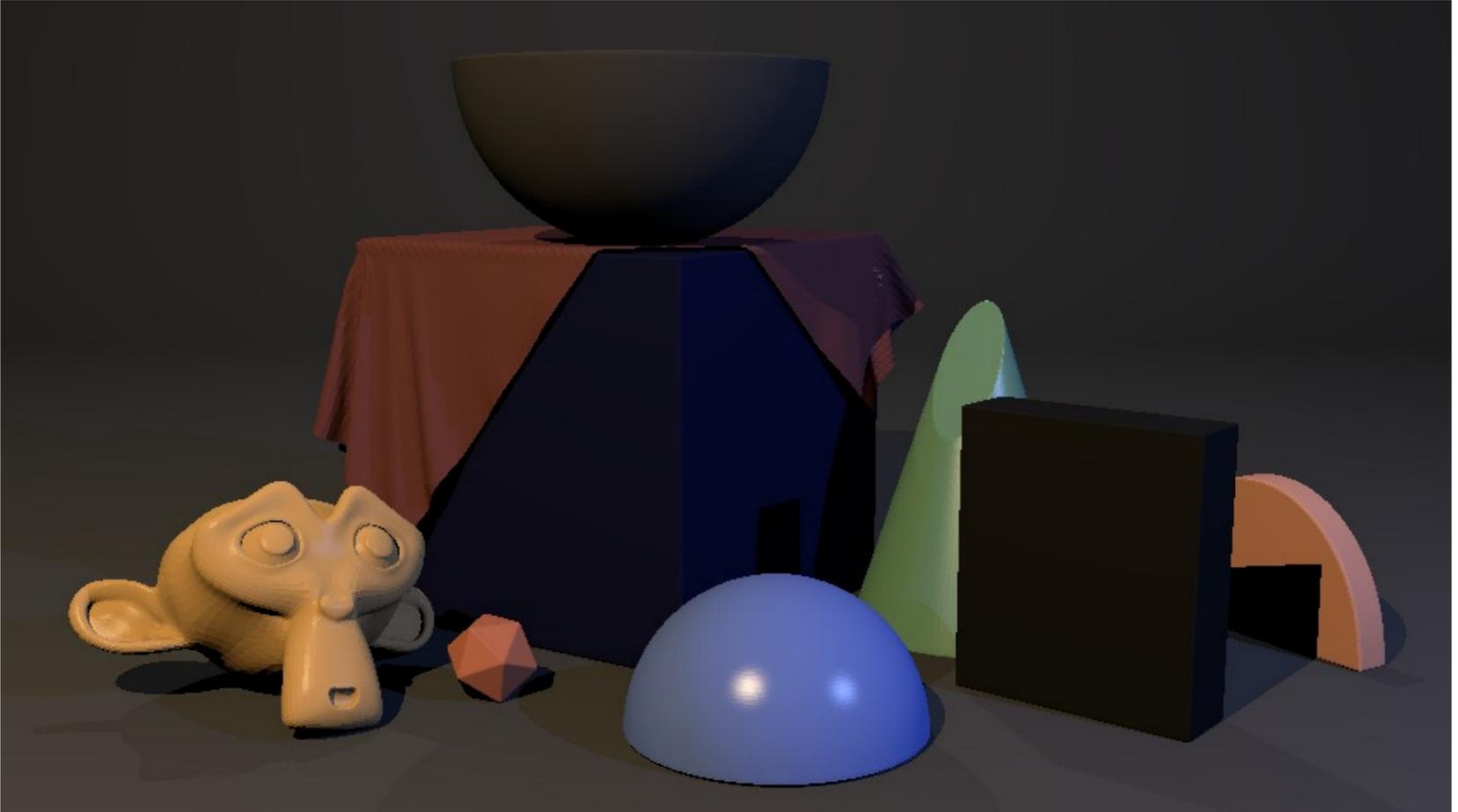
II.1. Blinn-Phong Model

With only the diffuse component, the image should look like this.



Checkpoint 2.1: Diffuse Shading. (For self-check only, no need to save the image)

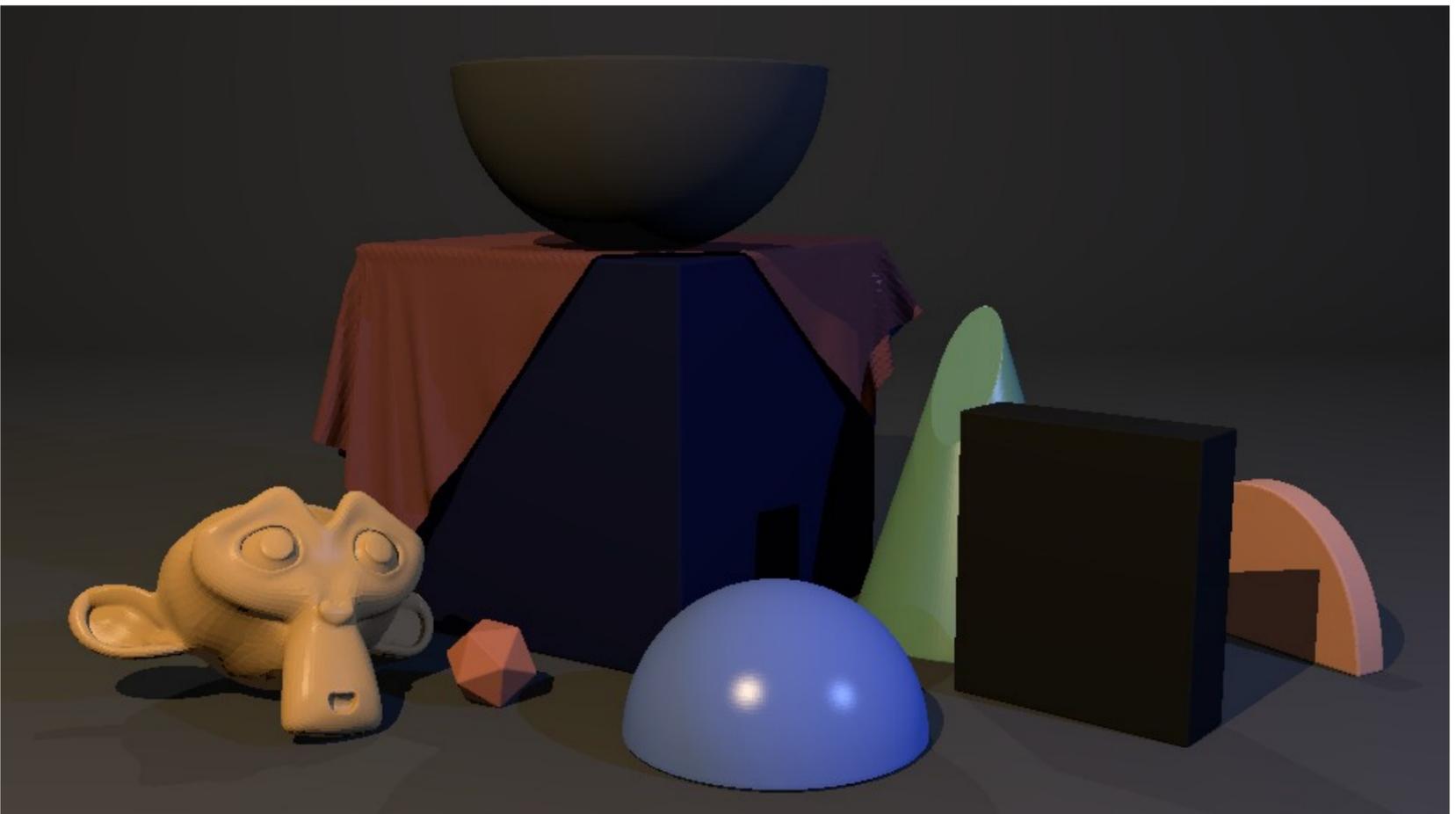
After adding the specular component, we have the full Blinn-Phong shading. Look for the colored specular light on the blue hemisphere in the middle.



Checkpoint 2.2: Save the image with Blinn-Phong Shading.

II.3. Ambient Light

The black area under the monkey and the shadow on the pink half-cylinder should brighten up. It is more obvious if you toggle between this and what you got from the last step.



Checkpoint 3: Save the image with ambient light.

II.4. Reflection

With recursive ray tracing, the hemisphere on the top should show a reflection of the whole scene.



Checkpoint 4: Save the image with reflection.

III.5. Fresnel

Since we're viewing the ground plane at a grazing angle, the reflection should be stronger.



Checkpoint 5: Save the image with fresnel.

III.6. Transmission

The black block should turn into glass, and the green truncated cone should show through it.



☐ **Checkpoint 6: Save the image with transmission.**

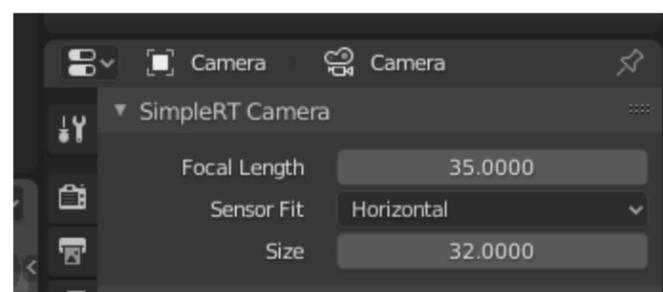
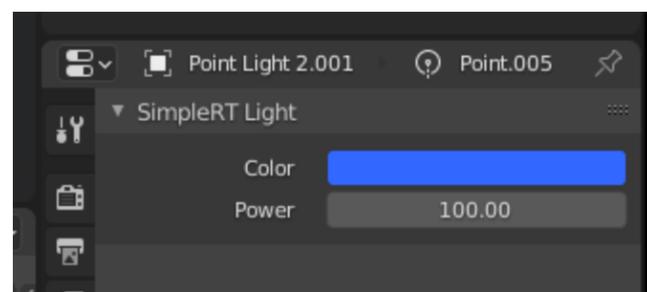
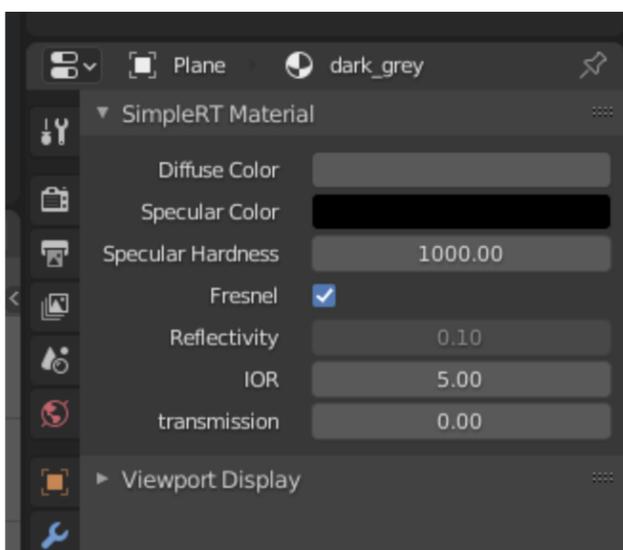
III. Play with camera, material, lighting

⚠ Remember to use the default scene and render settings for submission! Make a copy of the .blend file if you want to play with these settings.

Just as the render engines that come with Blender, we can change the camera, materials, and lights in our SimpleRT render engine. In the starter script, we have already set up the panels for materials/lights and hooked them up in the ray-tracing code.

For objects, we will use `SimpleRT Material`. If you select an object, then go to `Properties Editor → Material Tab`, you will see this panel. Here you can change all the settings, and they will be updated immediately. When you render, you will see the new settings applied. Same thing for lights and camera at `Properties Editor → Object Data Tab`. To get consistent results between the 3D viewport and the final render, please keep the `Sensor Fit` option to `Horizontal` for the camera.

If you want to dig really deep, the code for creating these custom panels are in the `simpleRT_UIpanels` script.



IV. Grading (5 pts total)

This assignment will be graded on the following requirements

Complete all the checkpoints (4 pts)

1. (0.5 pt) **Save the image with shadow ray.**
2. (0.5 pt) **Save the image with Blinn-Phong Shading.**
3. (0.5 pt) **Save the image with ambient light.**
4. (1 pt) **Save the image with reflection.**
5. (0.5 pt) **Save the image with fresnel.**
6. (1 pt) **Save the image with transmission.**

Quiz Question (1 pt)

The TAs will choose one of the following questions during the grading session. Please be prepared to give a one-minute answer with your partner.

1. What is one of the general practices to avoid self-intersection when generating a reflected ray on an object's surface?
2. In the ray tracing algorithm, how do we determine any additional influences on the color of a point on an object due to reflections and transmissions? Additionally, how do we know when to stop reflecting or transmitting to avoid stack overflows?
3. What is the difference between ambient and diffuse shading? Describe both algorithmic and visual differences.
4. How does the ray tracer use local surface normals to determine color/lighting of objects?
5. Why can metals reflect the color of surrounding objects and the environment from all viewing angles, but plastics cannot? Think about conductor and dielectrics material.
6. Explain Snell's Law.
7. Why is environment mapping useful?
8. In ray tracing algorithms, how can we determine if a point on the object is in shadow or not?