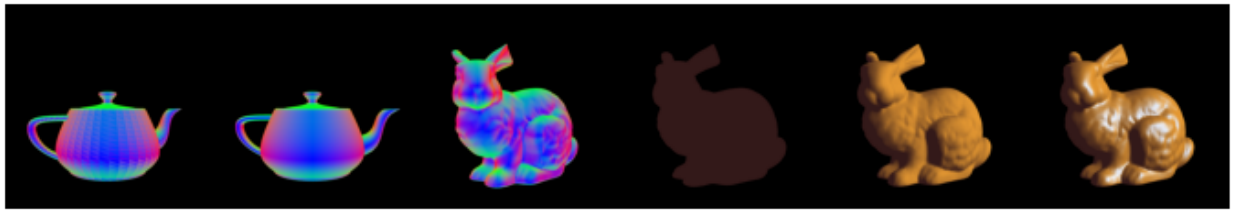
CS148 Homework 2

Homework Due: Jul 9nd at 11:59 PM PST
Quiz Date: Tuesday Jul 5th

1 Introduction

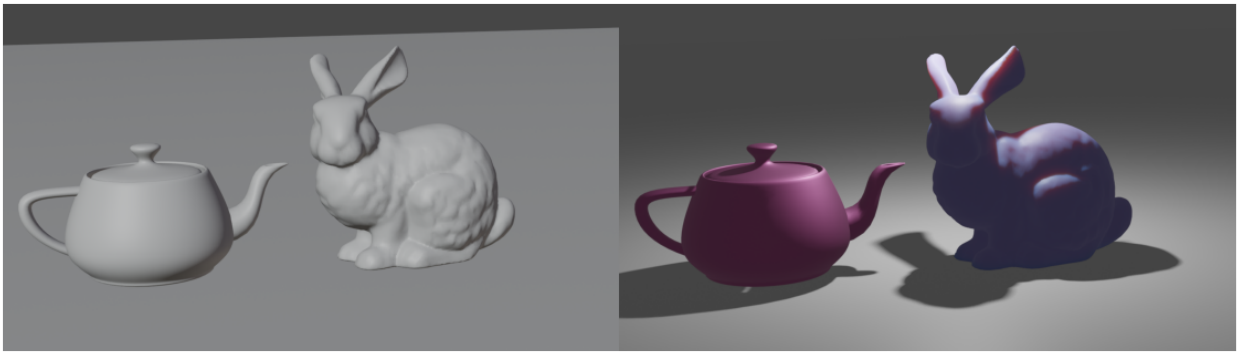
For the technical component of this assignment, you'll be working with GLSL and the rasterization pipeline via WebGL. (Note: you do not need any web dev experience to do this assignment!) Recall from lecture that one way to draw geometric shapes is a grid-based technique called rasterization, and that hardware in the form of GPUs were designed to do such computations at a speed that allowed real-time feedback, even back when computers were nowhere as powerful as they are now. OpenGL is a slightly outdated but very impactful rasterization framework that allows you to write code that directly interfaces with the GPU. (How cool is that?) Meanwhile, its web counterpart, WebGL, is still very relevant due to cross-platform compatibility, and is the backend powering a lot of web-based GUI applications and frameworks such as threeJS and Unity web games.



The technical component of this homework includes the following steps:

- (Warmup) Switch from flat shading to smooth shading
- (1 pt) Move computations from fragment shader to vertex shader
- (2 pts) Implement diffuse component of the phong shading model
- (1 pt) Implement specular component of the phong shading model
- (1 pt) Conceptual questions about the OpenGL coordinate system and color representation

Meanwhile, on the Blender side, you'll be testing out camera parameters and object materials!



More specifically, you will do the following in Blender:

- (1 pt) Add a single piece of geometry to the scene. Render at different focal lengths and discuss the qualitative changes.
- (2 pt) Pose 2 pieces of geometry on a ground plane. Adjust the camera so that both objects are within the render.
- (2 pt) Use the Eevee renderer and play around with the material parameters, so that the two objects you have look like they have distinctly different properties (ex. shiny vs marble-like).

Have fun!

2 Quiz questions

- Describe why we can see the color magenta even though magenta light does not exist on the spectrum of visible light.
- Describe the difference between phong shading and gouraud shading, and the pros and cons of each.
- Describe the difference between orthographic and perspective projection.
- Describe how a pinhole camera works.

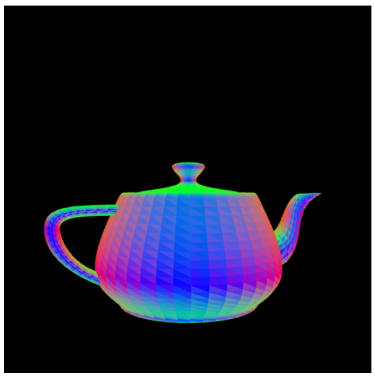
3 Homework download

Before we start, please download `HW2.zip`, save it in some course directory `$CS148_DIR` on your machine, and unzip the file in this directory.

4 WebGL

4.1 Setup

- Start up anaconda and the cs148 environment via `conda activate cs148_env`
- go into the HW2 directory, and launch a local web server via `python -m http.server –port 8889`.
- Now open your web browser (we've tested this on chrome and edge) in Incognito Mode, and go to `localhost:8889`. You should see a spinning teapot looking like this:



4.2 Warmup: Flat to Smooth shading

The only code you will need to touch in this assignment is `hw2.js` and `shaders/`, though feel free to read through the whole thing to understand what's happening. For those of you who've never done web development, the concept here is simple: The local server will launch our `index.html` file as a webpage at `localhost:8889`. `index.html` loads in `hw2.js`, `hw2.js` starts an instance of a webGL canvas, loads in a mesh, puts the mesh data in a format that WebGL can access, and compiles the shaders in `shaders/` based on what shader files we specify at the top of the `hw2.js`.

- Open up `hw2.js` in your favorite text editor. You can see that the vertex shader and fragment shader that gets passed into `hw2.js` is `shaders/starter_frag.js` and `shaders/starter_vert.js`.

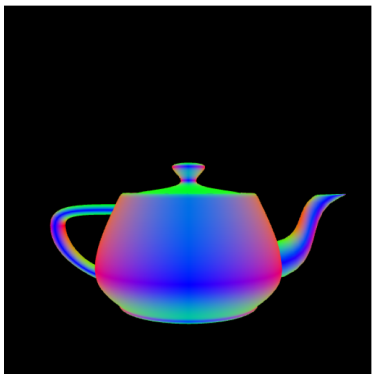
- Open up both `shaders/starter_frag.js` and `shaders/starter_vert.js` in the text editor. Recall that the output from the vertex shader (vertex-based attributes) goes into the fragment shader, and that the final color is the output from the fragment shader. The keywords `in` and `out` here denote the input/output parameters, where `uniform` defines additional inputs that are not dependent on geometry (hence "uniform" across all vertices and fragments.)

```
export default `#version 300 es
    precision highp float;
    precision highp int;

    in vec4 aVertexPosition;
    in vec4 aVertexNormal;

    uniform mat4 uMeshToWorldMatrix;
    uniform mat4 uMeshToWorldRotMatrix;
    uniform mat4 uWorldToClipSpaceMatrix;

    flat out vec4 vNormal;

void main(void) {
  // set vertex normal (to be passed to fragment shader)
  vNormal = uMeshToWorldRotMatrix * aVertexNormal;

  // set world space vertex position
  vec4 vPosition = uMeshToWorldMatrix * aVertexPosition;

  //openGL reads the clipping space coordinates from gl_Position
  gl_Position = uWorldToClipSpaceMatrix * vPosition;
}
`
```

```
export default `#version 300 es
    precision highp float;
    precision highp int;
    flat in vec4 vNormal;
    out vec4 fragColor;

    void main(void) {
        // compute color from interpolated normal
        vec4 vColor = vec4(normalize(abs(vNormal.xyz)),1.0);
        fragColor = vColor;
    }
`
```

Observe how the `flat out vNormal` in the vertex shader gets passed into the fragment shader as `flat in vNormal`. Now the `flat` modifier is one that tells the shader to "only use one vertex in each fragment to determine the entire fragment's color". So no interpolation is done in the fragment– it just uses a flat/uniform value from one of its vertices!

- With that knowledge, close your web browser. Remove the `flat` modifier from both the vertex and fragment shader, relaunch your browser, and you should now see a smooth and round teapot!



Note on closing your browser: unfortunately, due to modern web browsers doing aggressive caching, a simple refresh/reload of the page may not be enough to update the page with your code changes. We recommend starting with Incognito tabs to reduce this problem– if you expect to see changes but don't, close the tab and reload the page in a new tab. Thankfully, you should never have to restart your python server.

4.3 Moving computations from fragment to vertex shader

Recall that doing computations in the vertex shader is generally faster than doing so in the fragment shader. What we're currently doing in the starter shaders is that we're taking in the object space normals for each vertex, passing world space normals into the fragment shader, then for each pixel, we're computing the interpolated normal and converting it into RGB colors (defined between 0-1) by normalizing the per-component absolute value of the normal to have length 1.

For example, if the normal vector is $(3, -4, 0)$, we convert it to an RGB color by doing

$$\frac{abs((3,-4,0)}{||(3,4,0||} = \frac{(3,4,0)}{5} = (0.6, 0.8, 0.0)$$

(This is just one way of taking an arbitrary 3-dimensional vector and converting it to a vector with values between 0 and 1.)
(As a side note, color is actually represented as a 4 dimensional vector here, with the 4th channel corresponding to transparency.)
As a warmup step for the phong shader, modify the starter shaders so that the normalization and color conversion happens in the vertex shader instead of the fragment shader. Here's a hint of how you'd want to do this:

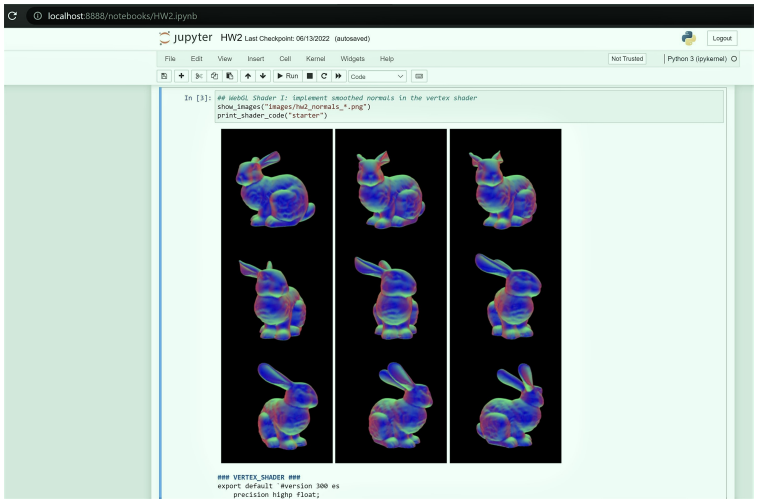
- All the code changes should happen in `shaders/starter_frag.js` and `shaders/starter_vert.js` and should only take less than 5 lines of code.

- You'll want to change `out vec4 vNormal` in the vertex shader to something like `out vec4 vColor`, and the same for `in vec4 vNormal` in the fragment shader!

After you've done this and verified it works (relaunch the browser tab and make sure you still see a spinning teapot), open up `hw2.js`.

- change `MAX_OUT_FRAMES` to 9.
- Change the obj file we're loading `teapot.obj` to `bunny.obj`
- Refresh/re-open your browser, and you should get 9 images downloaded to your default downloads directory.
- change `MAX_OUT_FRAMES` back to 0.

Action: Move the generated images into the `HW2/images` folder. At this point, you may want to launch the HW2 jupyter notebook with a separate anaconda prompt, and verify that the first 2 cells run successfully. You should see a grid of images and your shader code printed out.



4.4 Implementing the diffuse component

Now onto the phong shader! For both this section and the next, you should only have to change `hw2.js` and `shaders/phong_frag.js`. (There's nothing you need to change in the vertex shader!) If you switch the shaders at the top of `hw2.js` to use the phong shaders, via

```
import vsSource from "./shaders/phong_vert.js";
import fsSource from "./shaders/phong_frag.js";
```

you will be presented with one solid block of color (the ambient component). A couple of things you'll need to do to finish this section:

- Figure out how the ambientColor vector gets passed from `hw2.js` into the shaders. Mimic that logic and do the same for diffuseColor and specularColor so that you can refer to them in either the vertex/fragment shaders using the `in` keyword. (Hint: search for the TODO blocks in the code here).
- Refer to these variables in your fragment shader ex. by adding `uniform vec4 diffuseColor` in `shaders/phong_frag.js` right before the main function. Don't make any changes to main() just yet. Relaunch your browser and make sure there are no errors; you should still see a solid color block.
- Pass the world space vertex normal from the vertex shader into the fragment shader using `in` `out` keywords.
- Compute the diffuse component in the fragment shader, using lightPosition and vNormal. Refer to lecture slides here for the motivation and details for the diffuse component

$$D = k_d * max(0, \hat{N} \cdot \hat{L})$$

- Add the diffuse component to the final output color.

If you start up your browser, you should now see something like this:

Now, in order to complete this section, go back to the top of `hw2.js` and generate images via the following steps:

- change `MAX_OUT_FRAMES` to 9, and change `IMAGE_PREFIX` to `diffuse` .

- Refresh/re-open your browser, and you should get 9 images downloaded to your default downloads directory.
- change `MAX_OUT_FRAMES` back to 0.

Action: Move the generated images into the `HW2/images` folder. At this point, you may want to launch the HW2 jupyter notebook with a separate anaconda prompt, and verify that the next cell runs successfully. You should see a grid of images and your shader code printed out.

4.5 Implementing the specular component

After you've finished with the diffuse component, the specular component

$$S = k_s * max(0, \hat{V} \cdot \hat{R})^{\alpha}$$

should be easy! Again, refer to lecture for the details here. This should be less than 10 lines in your fragment shader. Note that α is currently hardcoded in the fragment shader as `float specularAlpha=0.6`, and you should only add the specular component to the final color if $\hat{N} \cdot \hat{L}$ from the previous section is greater than zero.
Important hint: We've setup this scene to have the world to camera transform to simply be the identity matrix. What this means is that the object to world transform already puts all our objects in the camera frame of reference, and in other words, the viewer/eye/pinhole position is **at the origin=(0.f,0.f,0.f)**. You will need this information to compute $\hat{(R)}$.

One you see the mesh looking like this, go back to the top of `hw2.js` and generate images via the following steps:

- change `MAX_OUT_FRAMES` to 9, and change `IMAGE_PREFIX` to `specular` .
- Refresh/re-open your browser, and you should get 9 images downloaded to your default downloads directory.
- change `MAX_OUT_FRAMES` back to 0.

Action: Move the generated images into the `HW2/images` folder. At this point, you may want to launch the HW2 jupyter notebook with a separate anaconda prompt, and verify that the next cell runs successfully. You should see a grid of images and your shader code printed out.

4.6 The WebGL coordinate system and the camera

The last section of the homework is more of a conceptual exercise. Recall that blender is a z-up system, which means that the upward direction corresponds to the positive z axis. However, OpenGL/WebGL and a big percentage of rendering software have the positive z-axis pointing towards the user (coming out of the screen, like the ghost in The Ring...) In particular, in the camera space, the z-axis is what is pointing inwards towards the camera. With that context, think about the questions in the jupyter notebook.

4.7 Resources

If WebGL and GLSL shaders are topics you want to look into further, Shadertoy is a great place to see some examples of other things you can achieve with it! The Phong shader and reflectance model is really the bread and butter of shaders, but there's a lot more you can do.

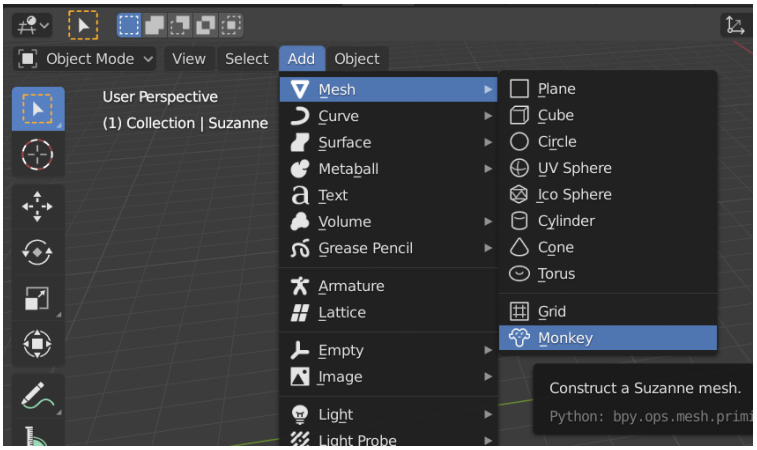
5 Blender Materials

For the blender part of the assignment, you will be working with cameras and playing around with materials! Note that the phong illumination/shading model you implemented just now is the basis of a lot of "default/basic" shaders in rasterization engines. The Workbench engine is the simplest and fastest rasterizer within Blender (with only one type of shader), which is the default viewport shading mode. The Eevee engine, while still being an OpenGL-based rasterizer (with additional bells and whistles ex. shadow mapping, soft shadows, and multipass rendering), is much more realistic while still being relatively fast. The slowest yet most versatile and photorealistic is the Cycles engine, the inbuilt "raytracer" (as opposed to rasterizer). We will not work with Cycles just yet, but you will learn more about how it works (and how to use it) in week 3's lectures and subsequent assignments.

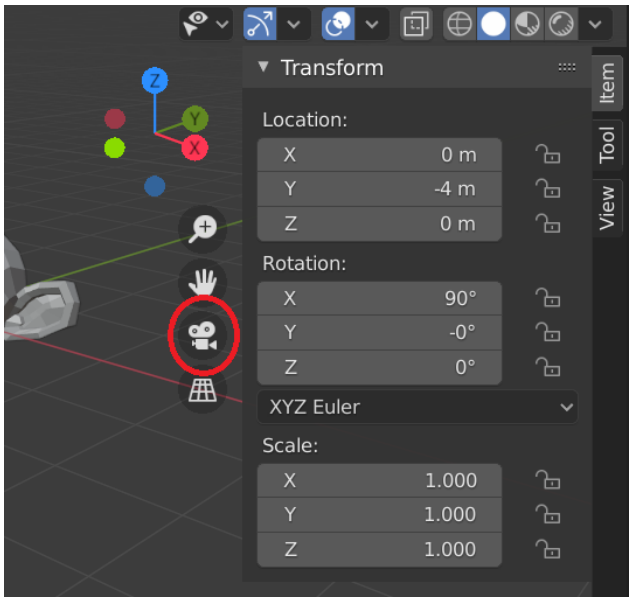
5.1 Cameras

We utilize camera models to project our 3D models onto a 2D image. In this section, we will play around with the perspective camera and understand the effect of focal length. This is exactly how dolly zoom (or vertigo effect), a famous camera effect, works.

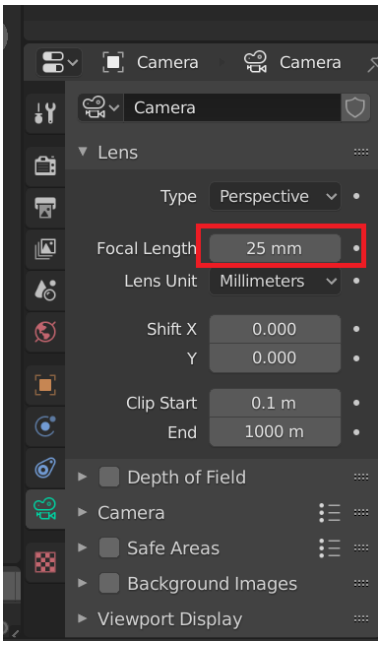
- First, we are going to set up the scene. Open a new Blender file, delete the cube. Add a monkey.



- Make sure the transformation sidebar is visible by toggling **View → Sidebar**, select the camera in the outliner (upper right of the interface), and set the camera location to $(0, -4, 0)$ and rotation to $(90, 0, 0)$. Press the camera icon within the viewport (circled in red) to go to the camera view.



- In the properties editor (lower left), change the focal length to 25mm. Render this out to **HW2/images/camera_25**. (If you've forgotten how to generate a render, review HW0! In particular, remember to set the start and end frames both to 1.)



- Change the camera location to $(0, -8, 0)$ and focal length to 50 mm. Render this out to **HW2/images/camera_50**.

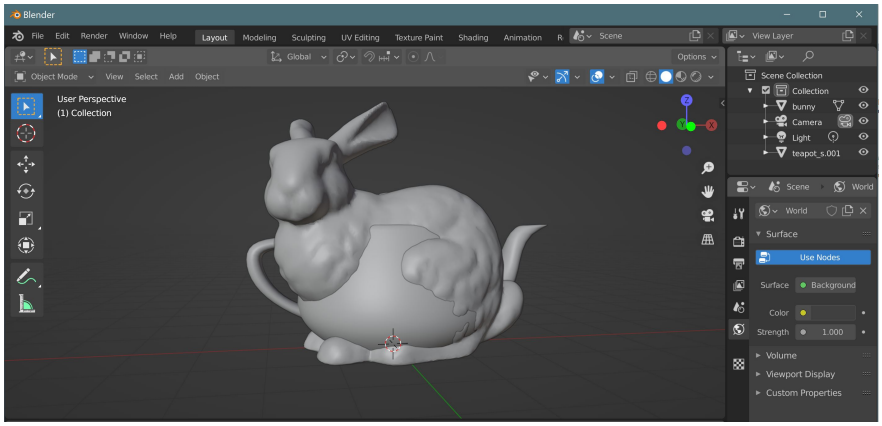
- Change the camera location to $(0, -16, 0)$ and focal length to 50 mm. Render this out to `HW2/images/camera_100`.

- **Action:** Compare these 3 images and discuss the effect of changing the focal length. Put your answers in the jupyter notebook. (You do not need to include the images in your writeup).

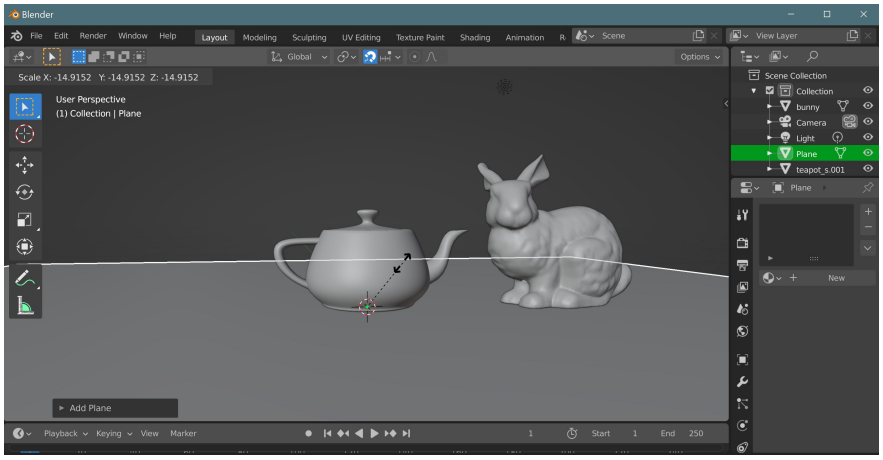
5.2 Eevee Materials: A fancier rasterizer

For this part of the assignment, you will start working with setting up a scene! Feel free to start a new scene or just delete the monkey from the previous part.

- **import two models**: one can be the bunny/teapot provided in this assignment in `HW2/assets` or downloaded from online, but **at least one model has to be something you've made**. For this second model, feel free to re-use the object you have from HW1 without changing it, though you might want to start building up your model collection for your final image!

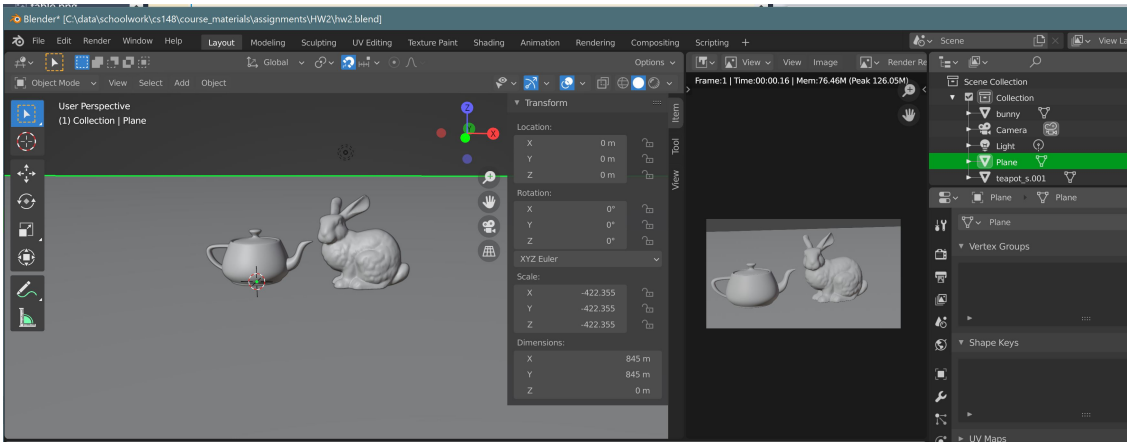


- **Create a ground plane and arrange your two models on top.** Go to `Add→Mesh→Plane`. Practice the transform shortcuts you learned from the HW1 (G for translation, R for rotate, S for scale, etc.) to arrange your objects on top of the ground plane. You may need to scale up the plane quite a bit depending on your model size.

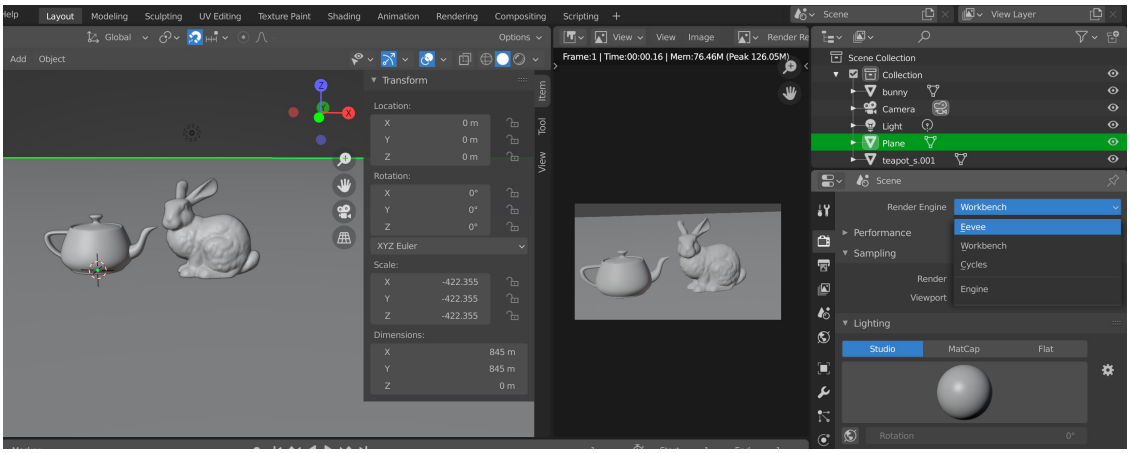


- **Change the camera view so that your objects are all fully contained within the render.** If you retrace the steps in the previous part of this assignment, you can specify the

camera transforms by explicitly setting it in the sidebar of your viewport. However, a slightly more intuitive way to iterate on camera angle is to rotate your scene to a position you are happy with in the viewport, and then click **View → Align View → Align Camera to Active View**. What this will do is it will try to set the camera so the render will match the viewport. Try rendering from here and tweak the camera slightly (via the rotion/translation tools) to your liking or until both objects are fully visible within the render and not cropped out or overly small.

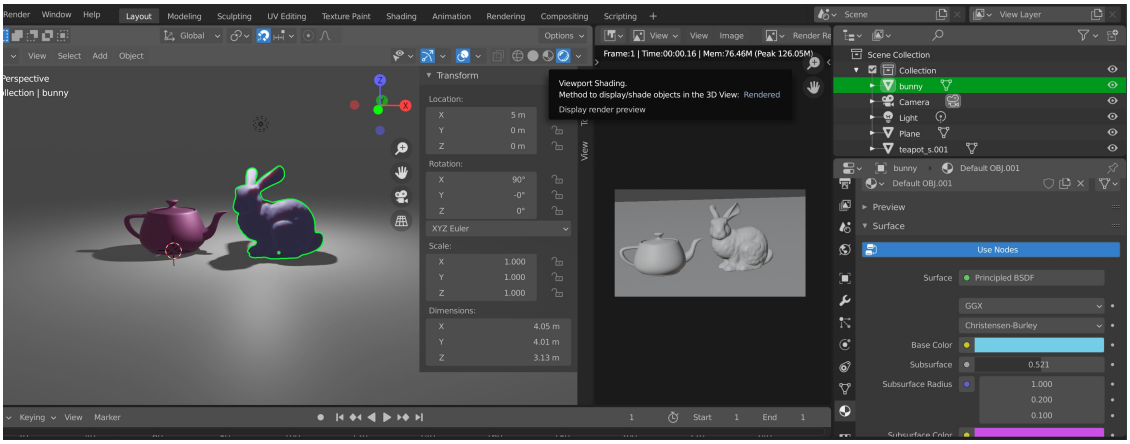


- **Switch the render engine to Eevee.** Go to the camera icon in the properties editor and switch your render engine to Eevee if it's not currently it already. Also make sure that the start and end frames for your renders (setting is under the image icon) is still both 1.

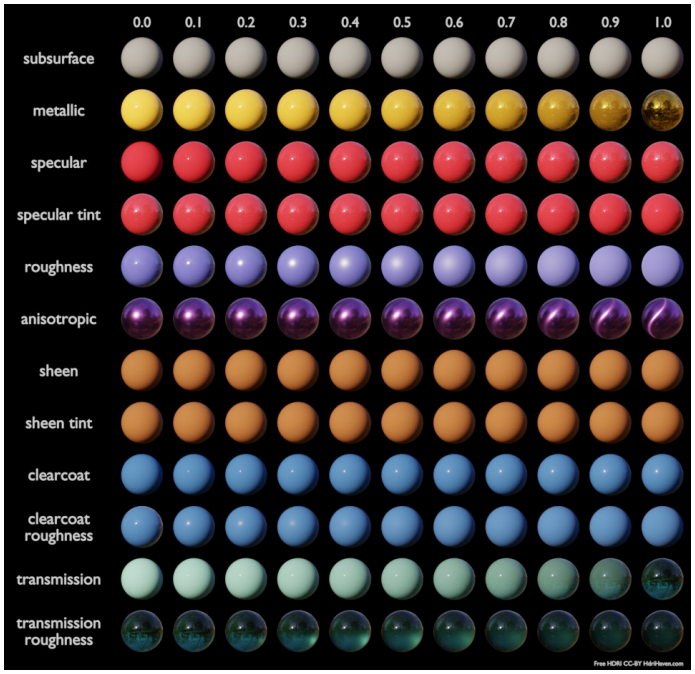


- **Switch the viewport shading mode and change the materials of your two models.** In the upper right corner of your viewport, there should be 4 sphere icons. Click on the rightmost one to switch to a full rendering mode. As you go from left to right, the viewport will become less and less responsive, but the shading gets more complex and close to what the final render looks like. **Tip:** you will generally want to stay in the solid and/or wire-frame modes (leftmost and second-leftmost spheres) as you work on object placement, only switching to the material and rendering modes when you're starting to work on materials and lighting.

In the properties editor, scroll to the bottom and find the materials icon (a sphere split into 4 quadrants.) Set the **Surface** type to Principled BSDF and play with the settings.



We'll talk about the parameters more in detail later, but the Eevee rasterizer's implementation of Principled BSDF is an OpenGL-based shader that is doing an approximation of a raytracing material model. It does have its limitations, where parameters such as transmission are difficult to implement in OpenGL and as such do not work nicely in the Eevee rasterizer.



Some parameters we recommend playing around with include: setting the subsurface value to a nonzero number, and setting the subsurface color to something different from the base color, and playing around with metallic parameters and roughness parameters.

- **Action:** Once you're happy with the way the materials look, write the render out to **HW2/images/eevee** (Again, refer to HW0 for a refresher) and make sure it is correctly displayed in your jupyter notebook.
- **For this assignment, don't worry about lighting! We'll work on that later.**

6 Checklist

When you submit your assignment to Gradescope, the submission should have the following:

1. 3x3 grid bunny renders of smooth normal shading, and your shader code that does gourad shading (in the vertex shader) instead of phong shading (in the fragment shader)
2. 3x3 grid bunny renders of the "ambient + diffuse" phong reflection model
3. 3x3 grid bunny renders of the "ambient + diffuse + specular" phong reflection model, and your shader code
4. Short response answers about WebGL and colors
5. Short response answers about cameras
6. Eevee render of 2 objects, at least one handmade. Both objects are fully within the image (not partially cropped) and have different material properties