CSE 167 (FA 2021) Homework 3

The goal of this homework is

- to understand scene graph and its traversal with a matrix stack;
- to implement lighting;
- to build and render a complex scene by yourself.

Download the skeleton code. You will be implementing

- `src/Scene.cpp`, specifically complete the code for `void Scene::draw()`
- `shaders/lighting.frag`.

For the last part of this assignment, you will make a backup for `src/Scene.inl`, and modify the code to build your using your creativity.

3.1 Before implementation

Make sure you can compile and run it. On Mac and Linux, run

```
make;
./SceneViewer
```

On Windows, compile and run in Visual Studio.

Pressing the arrow keys and A, Z will allow us to rotate and zoom the camera. Pressing L will switch between the normal shading (like the HW2 style shading) and the new lighting shading. Similar to HW2, pressing the spacebar will produce some screenshots.

Before any implementation, you can see a cube, a bunny and a teapot all smashed together. The arrow keys and the A, Z should work. Pressing L will turn the geometries pitch-black.

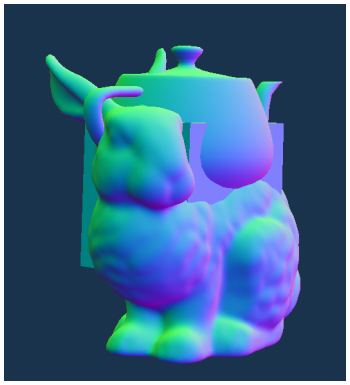


Figure 1 Before any implementation, we see a bunny, a cube, and a teapot all smashed together. This is because they are not rendered with the correct modelview matrix.

Read `src/Scene.inl`, which has the scene description implemented in `Scene::init()`. With Week4's lectures, you should be able to understand every line and understand that it corresponds to the scene graph shown in Figure 2.

You want to read this scene construction file so that you understand how each matrix/node is stored and accessed. Note that there are many -> and . operators to access members from a pointer or from an object. Make sure that the C++ code makes sense to you early on.

3.2 Step1: Matrix stack (3 pts)

The first task is to complete the implementation of `void Scene::draw()` in the file `src/Scene.cpp`. A depth-first search algorithm has already been there, and therefore the program is able

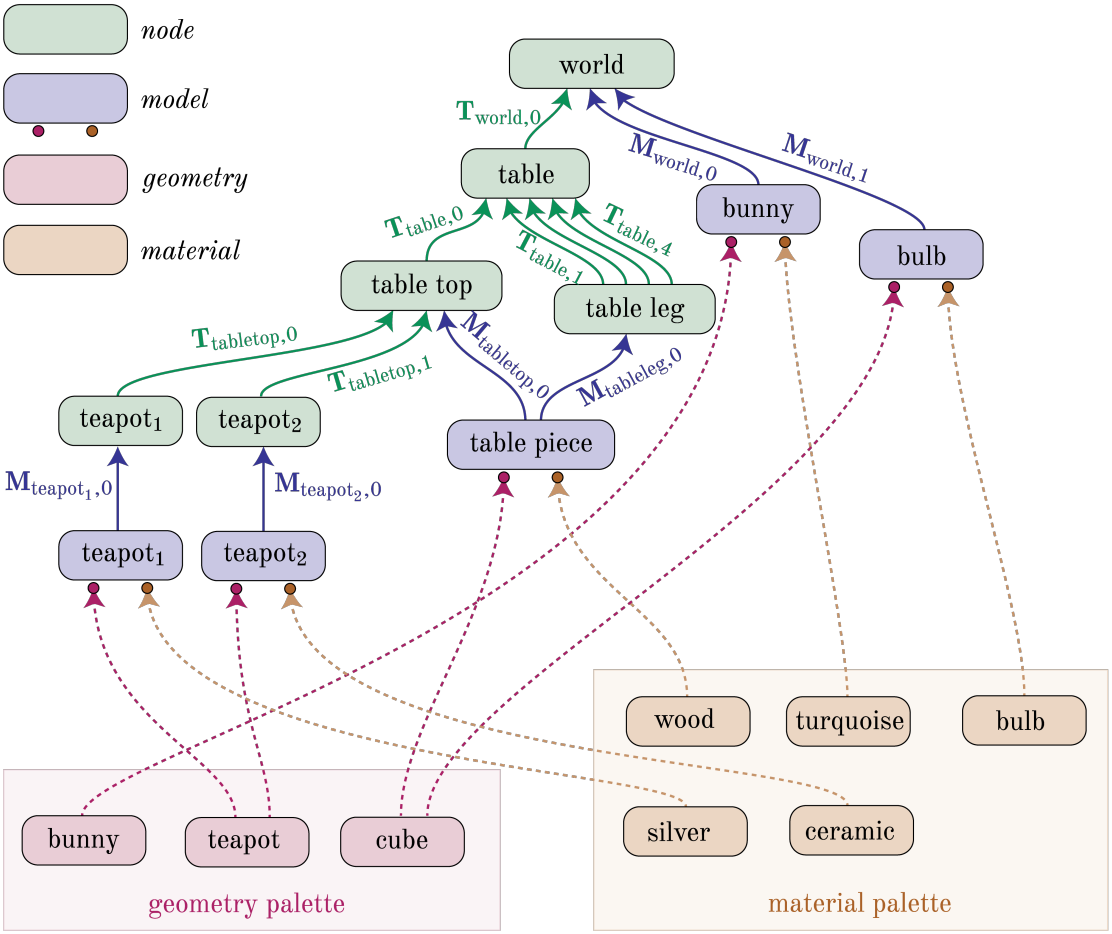


Figure 2 Scene graph for HW3. Here $\mathbf{T}_{\text{node},i}$ and $\mathbf{M}_{\text{node},i}$ are the ith transformation matrices stored in `childtransforms` and `modeltransforms` respectively.

to draw every object in the scene. The only thing that is missing here is to keep a matrix stack updated alongside the depth-first search. Chapter 9 of the lecture note can be helpful.

If this part is correctly implemented, the scene should become two teapots sitting on a desk next to a bunny (Figure 3).

For this part, you will submit the file `src/Scene.cpp` to gradescope.

3.3 Step2: Lighting (10 pts)

The next step is to implement the lighting in `shaders/lighting.frag`. In particular, we are performing a per-pixel shading computation (Phong shading). Let $\mathbf{L}_j \in \mathbb{R}^3$ (or $\mathbb{R}^4$, but the last component is dummy) denote the light color of the j-th light. Each coefficient such as $\mathbf{C}_{\text{ambient}}$, $\mathbf{C}_{\text{diffuse}}$, *etc.*, is also an $\mathbb{R}^3$ (or $\mathbb{R}^4$, ignore the last component) that represents the transmittance/reflectance for each of the red/green/blue channel. We can component-wise multiply a coefficient and a light color, $\mathbf{CL}$, to get the resulting light after being filtered by $\mathbf{C}$.

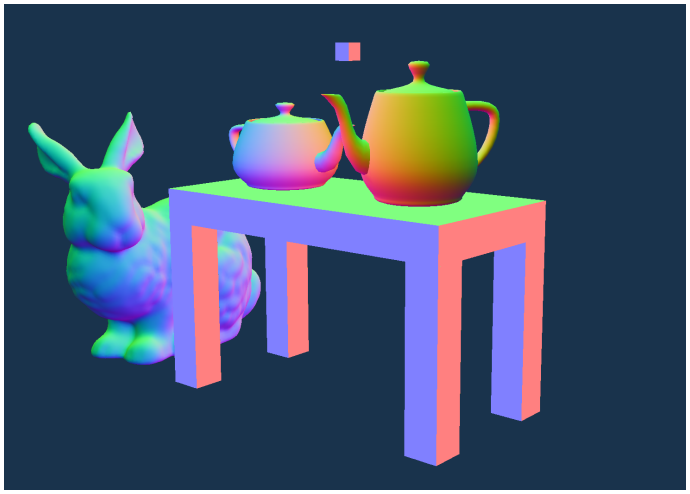


Figure 3

The final reflected color $\mathbf{R}$ shown in our pixel follows the Blinn–Phong shading equation:

$$\mathbf{R} = \mathbf{E} + \sum_{j=1}^{\text{num of lights}} \left(\mathbf{C}_{\text{ambient}} + \mathbf{C}_{\text{diffuse}} \max(\mathbf{n} \cdot \mathbf{l}_j, 0) + \mathbf{C}_{\text{specular}} \max(\mathbf{n} \cdot \mathbf{h}_j, 0)^{\sigma} \right) \mathbf{L}_j. \tag{1}$$

Here, $\mathbf{E}$ is the self-emission of the material, $\sigma \in \mathbb{R}$ is the shininess coefficient, and $\mathbf{C}_{\text{ambient}}$, $\mathbf{C}_{\text{diffuse}}$, $\mathbf{C}_{\text{specular}}$ are the ambient, diffuse and specular coefficients of the material. The vectors $\mathbf{n}, \mathbf{l}_j, \mathbf{h}_j$ have length 1. The vector $\mathbf{n}$ is the surface normal, $\mathbf{l}_j$ is the direction to the light, $\mathbf{h}_j$ is the half-way direction between the direction to the viewer and the direction to the light

$$\mathbf{h}_j = \textsc{Normalize}(\mathbf{v} + \mathbf{l}_j), \quad \mathbf{v} = \text{unit vector pointing towards the viewer.} \tag{2}$$

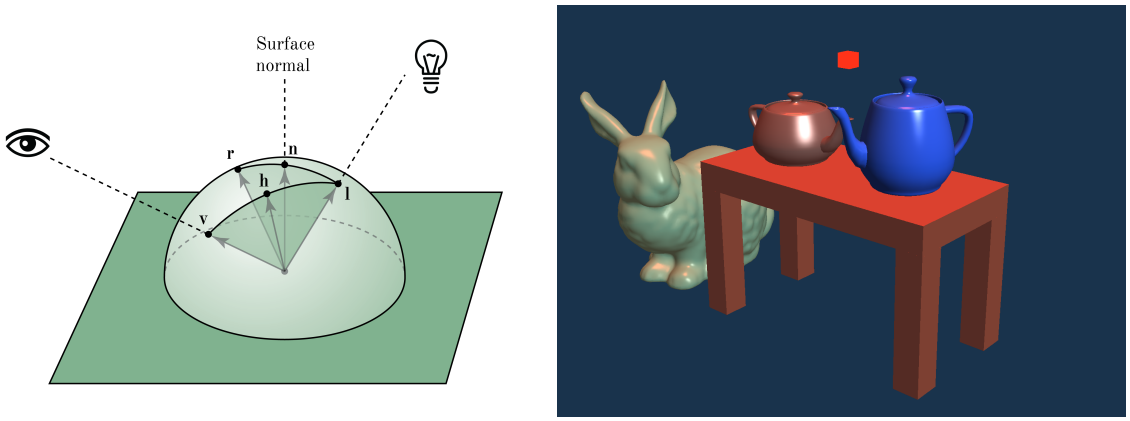
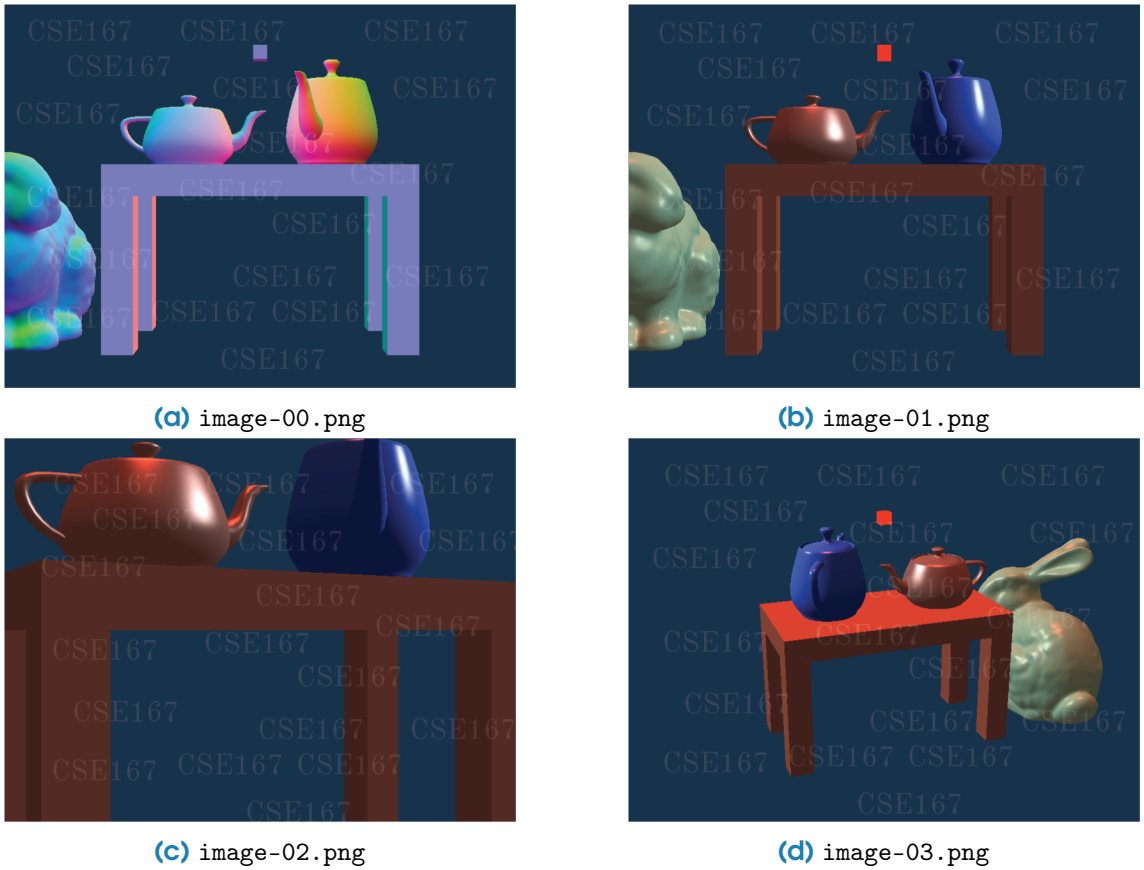


Figure 4 Left: Blinn–Phong formula; when the half-way vector $\mathbf{h}$ is close to the surface normal $\mathbf{n}$, the view direction is close to mirror reflection direction. Right: the result of lighting.

If this step is corrected implemented, you can press spacebar and obtain Figure 5.

For this part, you will submit to gradescope the file `shaders/lighting.frag`, and the 4 images from pressing the spacebar.



(a) `image-00.png` (b) `image-01.png`

(c) `image-02.png` (d) `image-03.png`

Figure 5 Reference images (with watermark).

3.4 Step3: Build your own scene (2 pts)

Make a copy of `src/Scene.inl` (as the backup) and you can come up with your own scene. That is, you have the freedom to define your own material, your own placement of objects, *etc*.

For this part, you will submit to gradescope your rendering and the file `src/Scene.inl`. You are welcome to define more inherited classes of Geometry if you want to. Don't worry about that these files are not included in the submission. We just need to look at `src/Scene.inl`, your rendering, and make sure you had fun.

3.5 Competition (+2 pts for winners)

You have the option of participating in a competition for your rendering of the scene you built. You and your classmates can have 3 votes for their favorite renderings. A small amount of bonus points will be added for the top 10 winners.

3.6 Hints

For the lighting formula, make sure that all positions (*e.g.* lights and the point position on the surface) and directional vectors (normal, directions to the lights and to the viewer) are referencing the same coordinate frame. It can be the world frame, or the camera frame.

Make sure to transform the normal vector from the model's coordinate to the world(or camera) coordinate. Check out the last few slides of "Transformation Recap" for the transformation rule for the normal vectors.

Each position is represented as a 4D vector. The 3D location it corresponds to is defined by the *xyz* component divided by the *w* component (dehomogenization). But, do note that one of the light is placed at infinity (the light position has a vanishing *w* coordinate), which will lead to division by zero if you simply convert all 4D vectors into 3D vectors.

To avoid the division by *w*, think of what you really need and try to work around it. For example, suppose you want to find the unit direction $\mathbf{d}$ pointing from a point with position $\mathbf{p} \in \mathbb{R}^4$ to $\mathbf{q} \in \mathbb{R}^4$. Naively, we would compute

$$\mathbf{p}_{3D} = \frac{\mathbf{p}.xyz}{\mathbf{p}.w}, \quad \mathbf{q}_{3D} = \frac{\mathbf{q}.xyz}{\mathbf{q}.w}, \tag{3}$$

$$\mathbf{d} = \textsc{Normalize}\left(\mathbf{q}_{3D} - \mathbf{p}_{3D}\right) = \textsc{Normalize}\left(\frac{\mathbf{q}.xyz}{\mathbf{q}.w} - \frac{\mathbf{p}.xyz}{\mathbf{p}.w}\right) \tag{4}$$

which has the danger of division by zero. To work around the problem, we notice

$$\textsc{Normalize}\left(\frac{\mathbf{q}.xyz}{\mathbf{q}.w} - \frac{\mathbf{p}.xyz}{\mathbf{p}.w}\right) = \textsc{Normalize}\left(\mathbf{p}.w\,\mathbf{q}.xyz - \mathbf{q}.w\,\mathbf{p}.xyz\right) \tag{5}$$

by completing a common denominator and then throw away the denominator since we can remove any scaling factor in the normalization function. The new alternative formula is stable even if $\mathbf{q}.w \to 0$.