Homework 2: Lighting, Shading, and GLSL

EE267 Virtual Reality 2025

Due: 04/17/2025, 11:59pm

Instruction

Students should use JavaScript for this assignment, building on top of the provided starter code found on the course webpage. We recommend using the Chrome browser for debugging purposes (using the console and built-in debugger). Make sure hardware acceleration is turned on in the advanced settings in Chrome. Other browsers might work too, but will not be supported by the teaching staff in labs, piazza, and office hours.

Each homework will have a theoretical and programming part. The theoretical part of the homework is to be done individually, while the programming part can be worked on in groups of up to two. If you work in a group, make sure to acknowledge your team member when submitting on Gradescope. You can change your teams from assignment to assignment. Teams will share handed-out hardware (later on in the course).

Homeworks are to be submitted on Gradescope (sign-up code: **D3PYB3**). You will be asked to submit both a PDF containing all of your answers, plots, and insights in a **single PDF** as well as a zip of your code (more on this later). The code can be submitted as a group on Gradescope, but each student must submit their own PDF. Submit the PDF to the Gradescope submission titled *Homework 2: Lighting, Shading, and GLSL* and the zip of the code to *Homework 2: Code Submission*. For grading purposes, we include placeholders for the coding questions in the PDF submission; select any page of the submission for these.

You can find starter code for the homework on our github repository (see course website for the link). Please download and use this starter before starting to work on the programming part of this assignment.

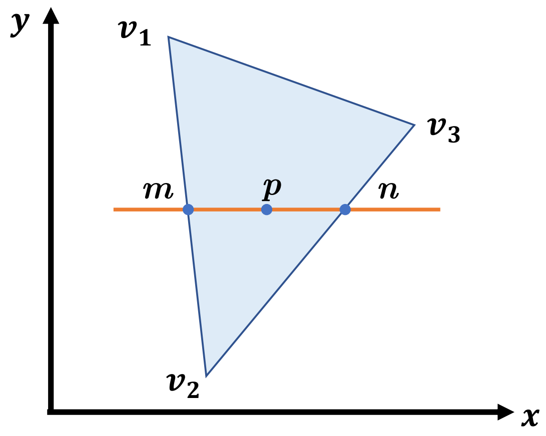
When zipping your code, make sure to zip up the directory for the current homework. For example, if zipping up your code for Homework 1, find `render.html`, go up one directory level, and then zip up the entire `homework1` directory. Your submission should only have the files that were provided to you. Do not modify the names or locations of the files in the directory in any way.

Please complete this week's lab and watch the video before you start to work on the programming part of this homework.

1 Theoretical Part

1.1 Simple Scanline Interpolation (*5pts*)

Given the polygon shown in the first figure below with vertices $\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}$ with coordinates $(x_i,\ y_i)$ for $i = 1, 2, 3$ and intensities at each vertex I_1, I_2, I_3, calculate the intensity at point $\mathbf{p}$ at $(x,\ y)$. Assume the coordinates of $\mathbf{m}$ and $\mathbf{n}$ are (x_m, y_m) and (x_n, y_n), respectively.



1.2 Phong Lighting (*30pts*)

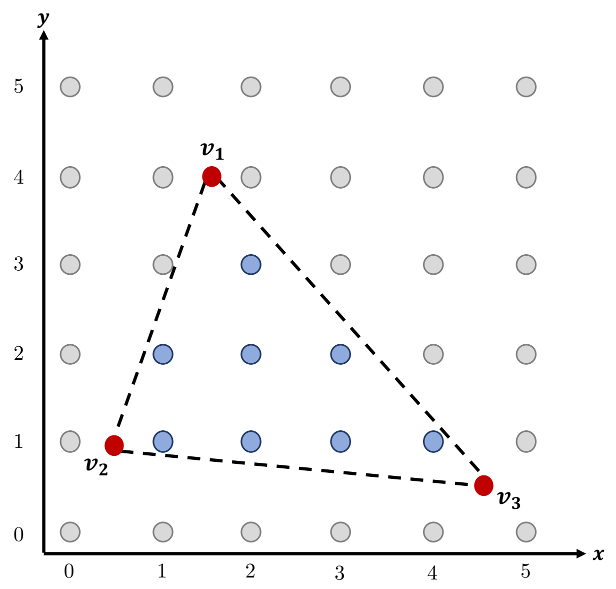
In this problem, we are going to "reverse transform" vertices from window space to view space where lighting calculations actually happen, and assign colors to each vertex based on the Phong Lighting model. As illustrated in the figure below, suppose we have a screen window with resolution of 6×6 pixels, both x and y components of pixel coordinates range from 0 to 5. The red points $\mathbf{v_1}$, $\mathbf{v_2}$ and $\mathbf{v_3}$ are vertices in window space while the blue points represent the set of fragments that were determined by the rasterizer to lie within the primitive (i.e. the triangle) spanned by the three vertices. You can think of fragments as a regular grid of pixels, each associated with a number of attributes such as 2D position, RGB color, normal, depth, alpha value etc. The rasterizer determines which fragments are inside a primitive and it interpolates vertex attributes such that each fragment inside the primitive receives a set of interpolated attributes. Refer to the lecture slides or Marschner's textbook if you are still confused. It is important to understand the concept of the rasterizer before moving on.

Given parameters of the three vertices:

- **vertex coordinates in window space:** $\mathbf{v_1} = (1.5, 4)$, $\mathbf{v_2} = (0.5, 1)$, $\mathbf{v_3} = (4.5, 0.5)$
- **depth values in window space with range** $[0,\ 1]$**:** $z_1 = 0.7$, $z_2 = 0.5$, $z_3 = 0.3$
- **normals in view space:** $\mathbf{n}_1 = (-\frac{2}{3},\ \frac{2}{3},\ \frac{1}{3})^T$, $\mathbf{n}_2 = (-\frac{2}{3},\ -\frac{2}{3},\ \frac{1}{3})^T$, $\mathbf{n}_3 = (\frac{2}{3},\ -\frac{2}{3},\ \frac{1}{3})^T$

(i) Compute 3D coordinates of all three vertices in view space, given the following parameters: (*15pts*)

- $aspect = \frac{\text{width}}{\text{height}} = 1$
- $fovy = 90^\circ$
- $zNear = 2$, $zFar = 22$



the projection matrix can be constructed as

$$M_{proj} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{6}{5} & -\frac{22}{5} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

and you may use this matrix in your calculation.

Hint: If you have a vector in NDC coordinates, you need to transform it into clip space by multiplying $\mathbf{v}_{ndc}$ by w_{clip} and plugging in w_{clip} as the fourth element. The problem is that we do not know w_{clip} directly. We can compute it however, from z_{ndc} and certain portions of the projection matrix. If we assume that the projection matrix has the following structure,

$$M_{proj} = \begin{pmatrix} \star & \star & \star & \star \\ \star & \star & \star & \star \\ 0 & 0 & T_1 & T_2 \\ 0 & 0 & E_1 & 0 \end{pmatrix}$$

where $\star$ denotes trivial elements that are not being used. Since we know

$$z_{clip} = T_1 \times z_{view} + T_2$$

$$w_{clip} = E_1 \times z_{view}$$

$$z_{ndc} = z_{clip}/w_{clip}$$

Then we can find w_{clip} as follows

$$w_{clip} = \frac{T_2}{z_{ndc} - \frac{T_1}{E_1}}$$

The derivation is not covered here but you are encouraged to derive it by yourself; this will not be graded.

(ii) Compute the color of each vertex using the Phong lighting model given the following conditions: (*10pts*)

- point light source located at $\mathbf{l} = (5, 5, 5)$ in view space with RGB color $\mathbf{I} = (10,\ 10,\ 10)^T$
- diffuse material properties: $\mathbf{m_1} = (1,\ 0,\ 0)^T$, $\mathbf{m_2} = (0,\ 1,\ 0)^T$, $\mathbf{m_3} = (0,\ 0,\ 1)^T$
- neglect specular and ambient components of lighting
- consider square distance falloff ($k_c = k_l = 0,\ k_q = 1$)

Note: In the Gouraud shading model (i.e., per-vertex lighting), the lighting calculations are done for each vertex similar to what you just calculated. Afterward, the rasterizer will interpolate the color values of the three vertices over the entire triangle (similar in spirit to what you just did). Remember that the Phong shading model (i.e., per-fragment lighting) uses the rasterizer to interpolate the vertex positions and also the normals over the triangle and then performs the lighting calculations per fragment.

In this particular case, Gouraud shading includes lighting calculations for the three vertices whereas Phong shading would require lighting calculations for all eight fragments that are inside this triangle. This would make per-fragment lighting significantly slower.

(iii) Oftentimes, per-fragment lighting involves more calculations than per-vertex lighting. Are there cases when this is not true? When specifically would Phong shading (i.e., per-fragment lighting) be faster than Gouraud shading (i.e., per-vertex lighting)? (*5pts*)

Programming Part PDF Deliverables

The following questions in the programming part ask you to provide written responses:

- 2.1.1.3 Attenuation Factor Observations
- 2.2.2.1 Gouraud vs Phong Shading Comparison

Make sure to append your responses to the end of the PDF submission.

2 Programming Part

When you first load up the starter code for this week's homework, you will see 5 dimly lit teapots (see Figure 1. The color would be different from the starter code.). They seem pretty bland and unrealistic. Well, that's because we haven't modeled the lighting yet, which is the point of this homework! Accurately modeling light interactions with materials can be complicated, but we also want to do it fast. Remember that a key part of VR/AR is keeping latency to a minimum. Therefore, we approximate the lighting calculation in ways to make them computationally efficient while still looking plausible. For this purpose, we use the Phong lighting model. You will be implementing this lighting model in various ways using GLSL shaders. By the end of the assignment you will have created a scene that looks like Figure 2. The color may not match because the figures use our old screenshots.

(You may see some warnings in the console. They are expected and should disappear after implementing all questions.)

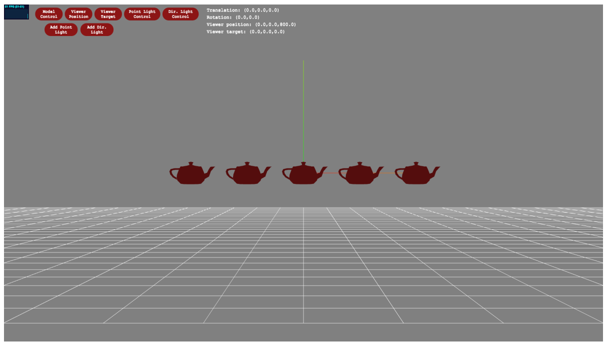


Figure 1: *teapots with only ambient light*

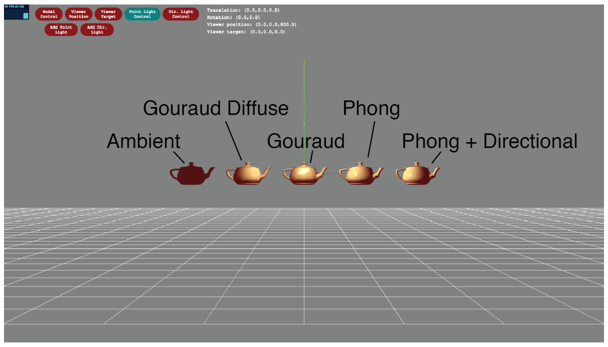


Figure 2: *teapots with lighting models*

We first need to approximate the light source itself. We typically want to approximate the color spectrum as three separate color channels RGB and we neglect global illumination. In this homework, we will model light as either a set of point sources (e.g., small light bulbs) or directional light sources (e.g., the sun or distant light sources), which will be defined by their position or direction, respectively, and color.

In the starter code, a point light is set up in the initial scene. A new point source is added with each click of the **Add Point Light** button and will appear as a new element in the `pointLights` struct array uniform variable in the shaders. You will need to access these structs when doing your computations. You can also move the light sources by clicking the Point Light Control button and dragging your mouse.

Remember that the color we perceive depends on both the light and the material the light bounces off. Therefore, we must also define the material properties. These material properties are model dependent, and you may have many different materials in a single scene. Material properties are passed to the shaders as uniform variables: `ambient`, `diffuse`, `specular`, and `shininess` parameters as a struct for each color channel. These uniform variables are defined in `standardRenderer.js`.

Another type of variable that you'll be using in the shaders is an attribute, which is a variable associated with each individual vertex. At a minimum the attributes must include vertex positions to be able to place a vertex, but can include things like normals, colors, texture coordinates, etc. In Three.js, the attribute variables such as vertex positions and normals are stored as properties of the `THREE.Geometry` class. If you are interested, you can view all of the teapot's attributes parsed to the shader program by typing `teapots[0].geometry.attributes` in console. These are the attributes you will be operating on in the shaders.

GLSL supports many built-in functions that you might find useful for this assignment like `dot()`, to compute a dot product, and `reflect()`, which computes the reflection of an incident ray off a surface with a normal. You can find all of the GLSL functions with short explanations here. Even though the shader files are JavaScript files, the GLSL code is embedded as string. Therefore, we encourage you to change the language setting to GLSL in your editor when you edit the shader JavaScript files for syntax highlighting. Before you begin, please update the `screenDiagonal` global variable in `render.js` with your physical screen's diagonal in inches.

Tip: You'll be implementing several shaders in GLSL, which can be difficult to debug. You can't use `print` statements, but you can instead use `gl_FragColor` to serve a similar purpose. For example, you can set `gl_FragColor.r` (via `vColor` if you're in the vertex shader) to the distance to the light source to see if it makes sense. Remember that any values will be clamped to the [0,1] range, so rescale the distance first; e.g., divide by 2000 to make "red" mean ≥2 m away.

After implementing the shader, you might see (*warning X3557: loop only executes for 1 iteration(s), forcing loop to unroll*) in the Chrome's console (you may not see it if you are on OSX). Even if you see it, you can ignore this warning. You may keep seeing it in the following homework too.

2.1 Gouraud Shading Model (*30pts*)

You will first implement Gouraud shading, which implements per-vertex lighting (in the vertex shader) and then lets the rasterizer interpolate the resulting colors to each fragment inside the primitives (i.e., triangles).

2.1.1 Diffuse Term (*15pts*)

2.1.1.1 Your first task will be to extend ambient lighting by the diffuse term in `vShaderGouraudDiffuse.js` and `fShaderGouraudDiffuse.js`. Your changes to these shaders will appear on the second teapot from the left. The light sources in the scene appear in the `pointLights` array of structs. For this task, assume only a single light source exists. Therefore, use the light source at the 0th index of the array.

In the shaders, extend the ambient lighting calculation by the diffuse term. Note that you will need the surface normal (given in object space) at the current vertex to do this calculation, which corresponds to the corresponding uniform variable in the provided vertex shader.

All lighting computations should be performed in the view coordinate system, such that the camera is in the origin looking into the negative z direction. That means, you need to transform the normal into the view coordinate system by multiplying it by the 3×3 normal matrix. You will find the matrices necessary for this already defined as uniforms in `vShaderGouraudDiffuse.js`. Also, assume that the light source position is given in world space. (*10pts*)

2.1.1.2 Attenuation Factor Add user-defined distance attenuation factors. Specifically, implement and compare the distance falloff ($k_c = 2$, $k_l = 0$, $k_q = 0.001$) with the constant attenuation ($k_c = 2$, $k_l = 0$, $k_q = 0$). For this purpose, you need to compute the distance between a light source and a vertex and calculate the appropriate distance falloff term. In `stateController.js`, the attenuation factors are defined as `THREE.Vector3(kc = 2, kl = 0, kq = 0)`. This gets mapped to the `attenuation` uniform variable in your shader files. (*3pts*)

Use the parameters ($k_c = 2$, $k_l = 0$, $kq = 0.001$) for the rest of the implementations (Gouraud and Phong shading with point light sources, but *NOT* for directional light sources).

2.1.1.3 Attenuation Factor Observations

Does enabling attenuation have a major impact on the appearance of the scene? If so, how so? Include your responses in the PDF submission. (*2pts*)

2.1.2 Specular Term (*5pts*)

Diffuse lighting is a step in the right direction, but it does not support any view-dependent lighting effects. In this task, extend the diffuse lighting model from the previous task to include specularities, following the Phong lighting model described in class. **Copy your diffuse lighting code into `vShaderGourad.js` first and then add the specular term.** This will show the specular highlights on the 3rd teapot, so that you can compare how diffuse-only and diffuse+specular lighting work. Extend your diffuse vertex shader from part A by the specular term. For this calculation you will need to compute the view vector from a 3D vertex position to the camera. Because you are operating in view coordinates, the camera will be in the origin, but you need to make sure to transform your vertex into the view coordinate system first. You might find the GLSL function `reflect()` to be useful for calculating the perfect reflector. Pay close attention to the direction to which the reflection is expected.

2.1.3 Multiple Light Sources (*10pts*)

Expecting there to only be one light source in a scene is a little unrealistic, right? In this task you'll extend the Gouraud shading model to an arbitrary number of light sources.

Clicking the **Add Point Light** in the browser will add additional point light sources to the scene. These additional light sources are represented in the `pointLights` array of `PointLight` structs in your shaders. Each element represents a separate light source. Remember that only a single color vector represents what is displayed on the screen at each pixel. This color, as per lecture slides, is the sum of the contributions from each light source interacting with each vertex (in Gouraud shading).

Extend all of the vertex shaders you have implemented so far by supporting multiple point light sources. Once complete, you should be able to add an arbitrary number of light sources and see the contribution of each light source on the rendering. There is no need to submit anything in the PDF. Just submit your code, and we will check its functionality.

2.2 Phong Shading Model (*20pts*)

Now that you've implemented Gouraud shading (i.e., per-vertex lighting), you'll next work on implementing Phong shading (i.e., per-fragment lighting) in the files `vShaderPhong.js` and `fShaderPhong.js`. For this purpose, you need to write a simple GLSL vertex shader that does all the 3D transformations and a separate fragment shader that does all the lighting calculations. As before, we will use the Phong lighting model and support multiple light sources.

2.2.1 Vertex Shader (*7pts*)

Write the vertex shader. There should be no lighting calculations in the vertex shader, only transformations. Transform the vertex and normal into the view coordinate system and write the results into the corresponding *varying* variables (`fragPosCam` and `normalCam`) as the output of the vertex shader. Attributes stored as *varying* variables will then automatically be interpolated by the rasterizer over the primitives and will be accessible for each fragment in the fragment shader.

2.2.2 Fragment Shader (*10pts*)

Write the fragment shader. The attributes defined as outputs of the vertex shader will now be accessible as inputs to the fragment shader. Note that the rasterizer did all the work for you and already interpolated these values to each fragment. Thus, in the fragment shader we have access to the interpolated 3D position and normal, both given in view space. It is possible that the interpolation process results in normals not being of unit length in the fragment shader, even though the per-vertex normals were normalized. Thus, make sure to re-normalize them in the fragment shader before doing the lighting calculations! You will perform the same lighting calculations that you already implemented in the vertex shader in Section 2.1 for each fragment here. Remember to support multiple light sources, just like in the Gouraud shading. The results of your Phong shading will appear on the second to right teapot.

2.2.2.1 Gouraud vs Phong Shading Comparison Compare Gouraud shading to Phong shading. How are they different? What are the benefits and downsides of each shading method? Specifically, comment on both quality of shading and computational load. You don't need to actually measure any runtimes here, just briefly discuss them theoretically. Report this comparison in your PDF submission. (*3pts*)

2.3 Directional Light Sources (*10pts*)

Until now we have been working with point light sources, which are sources with a given position in the world that radiate isotropically in all directions and where light rays attenuate with distance (via the attenuation factor). These are good light sources to model a light bulb or torch. When a light source is far away, however, the light rays hitting an object are close to being parallel. It looks like the rays come from the same direction regardless of the position of object and/or viewer. When a light source is modeled as being infinitely far away, all of its rays have the same direction and it is called a directional light source. A directional light source is independent of position, and you only need a direction vector to define the light source. This type of light source can be used to model, for example, the sun.

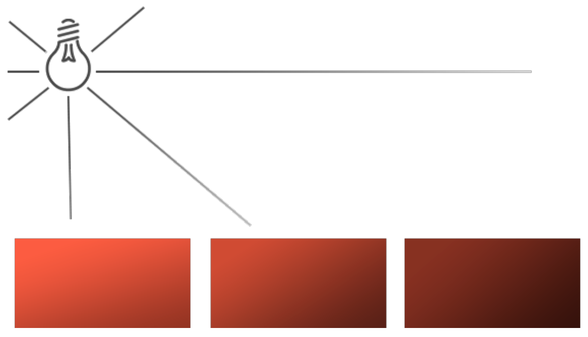


Figure 3: *point light source*

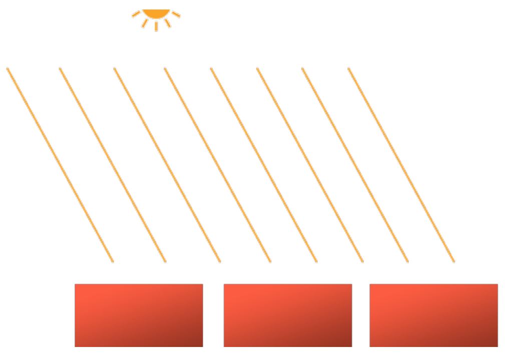


Figure 4: *directional light source*

Add support for an arbitrary amount of directional light sources in the `vShaderMultiPhong.js` and `fShaderMultiPhong.js` files. Copy your code from `vShaderPhong.js` and `fShaderPhong.js` to start. As with the point light sources, the initial scene has one directional light source, and clicking the **Add Dir. Light** button will generate additional directional light sources in the scene. The calculation is very similar to the point light source calculations from the previous task. Assume the provided direction of this light source is in world coordinates. If you see uncharacteristic lighting conditions with the directional light source, think about the direction of the directional light source vector and how it relates to the L vector in the Phong shading model we defined in class. Also, we typically do not assume that the intensity of light falls off for a directional light source, and therefore we do not apply an attenuation factor to the directional light source.

Once again we will check functionality of the implementation, and there is no need to submit anything in the PDF.

Questions?

First, Google it! It is a good habit to use the Internet to answer your question. For 99% of all your questions, the answer is easier found online than asking us. If you cannot figure it out this way, post on piazza or come to office hours.