Exercise 6

Computer Graphics - COMP.CE.430

Light Types and Participating Media

In this week's exercise, we will implement different types of light sources and model some simple volumetric effects. You should also take some time to make sure your light and view vector calculations are correct, as this will be important next week when we get to physically based shading.

Problem set

1. Add distance-based fog to your scene **(1p)**

   In this step, we will model light attenuation due to participating media. The assumptions we are making are that the density of the participating media is uniform across our scene and that the participating media covers the entire scene. In this exercise we will also only care about the attenuation that happens between the camera and the object being shaded: the attenuation between the light source and the object can be ignored.

   Light attenuation in participating media is caused by a combination of two factors: *absorption* and *outscattering*. Absorption models how much of the energy of the light is absorbed by the participating media. Outscattering models what percentage of the light along a view ray is scattered into other directions so that they no longer arrive into our camera along the view ray. Absorption and outscattering reduce the luminance of the incoming light independently, so we can combine them into an *extinction coefficient,* $\alpha_{ex} = \alpha_{ab} + \alpha_{sc}.$ You can think of the extinction coefficient as the density of the fog.
   The luminance of the incoming light falls off exponentially with the distance traveled in participating media, such that
   $L = L_0 \exp(-\alpha_{ex} d),$
   where *L* is the outgoing luminance, L_0 is the original luminance and *d* is the distance the light travels inside the participating media. The extinction coefficient α_{ex} is a value between 0 and 1, where 0 means no attenuation is happening and the incoming light is unaffected.

   Inscattering also affects the incoming light's luminance. Inscattering models external light that gets scattered in the direction of the camera, which has an additive effect. We will model this as a constant *fog color* C_{fog}, whose contribution increases exponentially with the distance.

The color and brightness of the light reaching the camera can be expressed as $C_{camera} = C_{shaded} f + C_{fog}(1 - f)$, where C_{shaded} is the shaded value computed in the fragment shader, C_{fog} is an RGB value representing the color and intensity of the participating media and *f* is a fog factor computed as $f = exp(-\alpha_{ex} d)$. You can use the exp function for computing the fog factor *f*.

It might be hard to visualize the effect of the fog in a vacuum. You can try adding a large cube with a negative scaling factor to surround your scene. A negative scaling factor will cause the normals to point inwards and only the inside faces of the cube will be visible if backface culling is enabled.

2. Add gamma correction to your shading **(1p)**

   Gamma correction is mandatory for any physically based shading, which we will be doing in next week's exercises. We have been ignoring it until now because our shading models have not been physically correct either. A lot of early 3D games ignored gamma correction completely (Quake 1-3 for example).

   Using a gamma value of $\gamma = 2.2$ is a close approximation for the sRGB color space. To apply gamma correction, raise your final shaded color to a power of $1/\gamma$ as the last step before outputting the color in the fragment shader.

3. Add height-dependent fog to your scene. **(1p)**

   In this exercise we will model non-uniform density fog, where the density depends on height. We can model the density variation with respect to height with an exponential function of the form $d(y) = a\exp(-by)$, where *a* is the fog's density at a height of 0, *b* is the falloff of the density with respect to height and *y* is the current height. This causes the density to decrease exponentially as the height increases. Solving the full equation involves solving an integral, but luckily it can be solved analytically and the final equation ends up fairly simple. For the full derivation and sample code, refer to this article by Inigo Quilez.

Bonus

1. Add a spotlight to your scene **(1p)**

   A spotlight is much like a point light, but with an added constraint based on the angle between the light vector and a direction vector that defines the direction of the spotlight. If you define the spotlight direction vector as a vector that points in the direction the spotlight is facing, you can compute the cosine of the angle between the spotlight direction and the light vector using the dot product:

```
float cos_angle = dot(spotlight_direction, -light_direction);
```

where light_direction is the normalized vector pointing from the fragment position to the spotlight position (the same vector you used when doing Phong shading).

You can compare the result to the cosine of some maximum angle. This will create a sharp edge. In order to create a smoother falloff, you will need to define an inner angle, below which the surface is fully lit, and an outer angle that defines where the light's contribution goes to zero, and interpolate between these. Several formulations for creating smooth falloffs exist. The one used by EA's Frostbite game engine is

$$t = \frac{\cos\theta_s - \cos\theta_o}{\cos\theta_i - \cos\theta_o},$$

$$f_{dir}(\mathbf{l}) = t^2$$

where θ_s is the angle between the spotlight direction vector and the reversed light vector $-\mathbf{l}$ to the surface, θ_o is the outer angle where the spotlight's contribution falls off to zero, and θ_i is the inner angle where the falloff begins. The variable *t* should be clamped to a range between 0 and 1. This formulation is also the one mentioned in the glTF spec. Multiply the light's intensity with the value of $f_{dir}(\mathbf{l})$ to create a smooth falloff. Remember that the inverse square law applies to spotlights as well.

A spotlight is much like a point light, but with an added constraint based on the angle between the light vector and a direction vector that defines the direction of the spotlight. If you define the spotlight direction vector as a vector that points in the direction the spotlight is facing, you can compute the cosine of the angle between the spotlight direction and the light vector using the dot product:

```
float cos_angle = dot(spotlight_direction, -light_direction);
```

where light_direction is the normalized vector pointing from the fragment position to the spotlight position (the same vector you used when doing Phong shading).

You can compare the result to the cosine of some maximum angle. This will create a sharp edge. In order to create a smoother falloff, you will need to define an inner angle, below which the surface is fully lit, and an outer angle that defines where the light's contribution goes to zero, and interpolate between these. Several formulations for creating smooth falloffs exist. The one used by EA's Frostbite game engine is

$$t = \frac{\cos\theta_s - \cos\theta_o}{\cos\theta_i - \cos\theta_o},$$

$$f_{dir}(\mathbf{l}) = t^2$$

where θ_s is the angle between the spotlight direction vector and the reversed light vector $-\mathbf{l}$ to the surface, θ_o is the outer angle where the spotlight's contribution falls off to zero, and θ_i is the inner angle where the falloff begins. The variable *t* should be clamped to a range between 0 and 1. This formulation is also the one mentioned in the glTF spec. Multiply the light's intensity with the value of $f_{dir}(\mathbf{l})$ to create a smooth falloff. Remember that the inverse square law applies to spotlights as well.

2. Add a line-shaped light source **(1p)**

You can define a line-shaped light source by defining two points that define the line segment. The light vector can be computed by finding the closest point to the fragment position on the line segment. The closest point can be computed by projecting the fragment position onto the line segment and clamping it to be between the two points that define the endpoints. If the line segment is defined by points *a* and *b*, you can compute the projection of a point *p* onto the line segment as

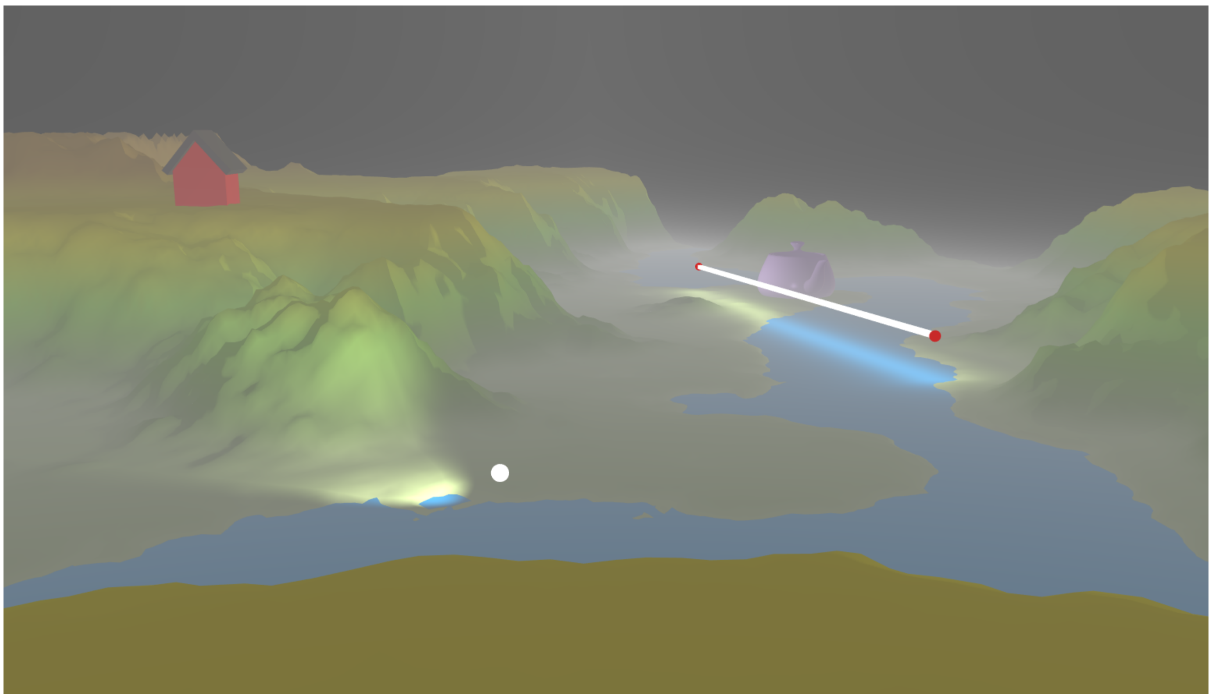
$$t = \frac{(\mathbf{b} - \mathbf{a}) \cdot (\mathbf{p} - \mathbf{a})}{(\mathbf{b} - \mathbf{a}) \cdot (\mathbf{b} - \mathbf{a})},$$

where *t* will have values between 0 and 1 when the point is within the line segment. If we clamp *t* to a range of 0 to 1, we can use this value to linearly interpolate between points *a* and *b* to compute the closest point on the line segment, and compute the light vector based on this point.

```
vec3 closest_point = mix(a, b, t);
```

Use this point to compute the light direction vector.

Example scene



You can find the model used for the terrain in this scene on Moodle.