Lighting

Lighting components

The required lighting information is stored within the game as a part of the ECT. The main components needed for the lighting setup are:

Component	Responsibility
Transformation	Used by lights and objects to determine their position.
ObjectMaterial	Stores the diffuse and specular values of the object.
Light	Stores the color of the light and the attenuation values for the shader

There are other components like the `VBA` and `TransformationMatrix` that are also required for drawing objects. However, these are not described in this file because they are not directly a part of the lighting model.

Shader layout

The lighting calculations are done in two stages. The first calculations are done in the vertex shader and then passed to the fragment shader where the final lighting value is determined.

Preparation

We need to do some preparation before drawing the actual object it self. The position and normal vector of a vertex is stored in the VBO and is passed to the vertex shader.

Model and scene specific data is stored in OpenGL uniform variables. The scene specific information like the light `positioning`, `color`, `attenuation` values and the camera position are uploaded at the start of the frame by the `PrepareFrameSystem`.

Vertex shader

The vertex shader calculates the vector S in our lighting calculations. It is a `vec3` value called `to_light` in our shaders.

Fragment shader

The pixel shader calculates the lighting with the formula we had in lecture 8. We added attenuation to our lighting system. The formal definition of attenuation is as follows:

> In physics, attenuation or, in some contexts, extinction is the gradual loss of flux intensity through a medium.

(Stolen from wikipedia)

This basically just means that a light has less effect on objects or in our case pixels if the further away they are from the light source. This enables us to have lights that don't have an infinite effect.

The attenuation can be set for every light or better say modified. The light has three attenuation factors: x, y, z. These factors are than used in the fragment shader in combination with the distance between the light and the actual pixel. The formula for the total light calculation can be seen here (With some awesome latex magic to make it look cool):

$$d := distance$$

$$f := shininess$$

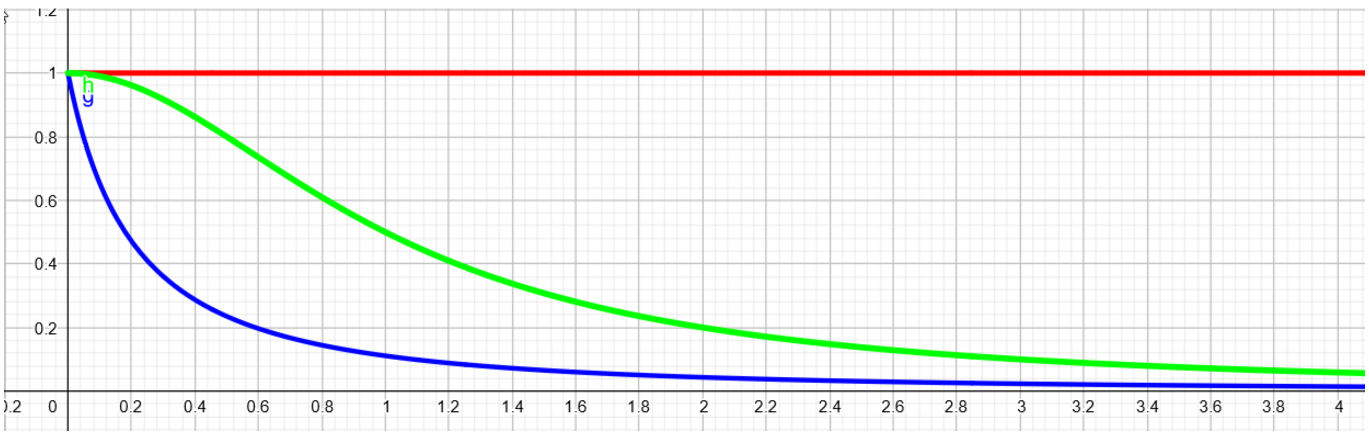
$$lambert = max\left(\frac{n \circ S}{|n| * |S|}, 0\right)$$

$$phong = max\left(\frac{n \circ h}{|n| * |h|}, 0\right)$$

$$attFactor = attenuation.x * d^2 + attenuation.y * d + attenuation.z$$

$$I = \sum\left(\left(I_d * \frac{material_d * lambert}{attFactor}\right) + \left(I_s * \frac{material_s * phong^f}{attFactor}\right)\right) + (I_{gaf} * material_a)$$

The calculation of the attenuation factor gives us a relatively wide range to play with the lighting effect while also not being to performance intensive. This following examples shows three example graphs for the attenuation:



- **Red**: attenuation = (x=0, y=0, z=1). This gives us an infinite light.
- **Green**: attenuation = (x=1, y=0, z=1).
- **Blue**: attenuation = (x=3, y=5, z=1).

Distribution of the Data / Lighting calculation

In this section we want to document the distribution of the lighting data and the reasoning for it

No.	What	Type	Shader
1	light_position	`vec3[]`	Vertex
2	light_count	`uint`	Vertex & Fragment
3	camera_position	`vec3`	Vertex
4	diffuse	`vec3`	Fragment

No.	What	Type	Shader
5	specular	`vec3`	Fragment
6	shininess	`uint`	Fragment
7	light_color	`vec3[]`	Fragment
8	light_attenuation	`vec3[]`	Fragment
9	global_ambient	`vec3`	Fragment

- (1 & 3) The explanation for the `light_position` and `camera_position` data is quite simple. We use them in the vertex shader because this is the shader where we also have access to the vertex position. Calculation the `s` vector (`to_light` in our case) for every vertex saves us some performance in the fragment shader
- (2) The light count is a value that is only used for optimization. It allows us to exit the light calculation loops a bit early if there are less lights than the `MAX_LIGHT_COUNT` macro indicates. (Note: GLSL requires a static loop end. This means that we can just loop from 0 to `light_count`. We still save performance by only doing an if check in the code)
- (SELECT * FROM TABLE) The rest comes down to accurate pixel lighting. I thought about calculating `lambert`, `phong` and other values in the fragment shader. I came to the conclusion that this wouldn't be a wise choice due to the following reasons:
    - The smaller reason but the performance gain would be minimal to actually negative. The fragment shader is executed before clipping. This means that we would therefore due these calculations for a lot of vertices that will never be drawn.
    - The second and main reason is a resulting inaccuracy. OpenGL, DirectX and other frameworks use interpolation to translate from the vertex results to the fragment input. This means that the result of the vertex shader decides which values can arrive in the fragment shader. The following image shows an example. The lighting values for `lambert` and `phong` would return a low value at P1 and P2. The interpolation would therefore also produce a low value for the point C even if it is basically in the perfect position 😃

```

                         --O--
 ^                        /|\                          ^
 |                                                     |
 |                                                     |
#######################################################
P1                         C                          P2
```

Multiple light setup

We have multiple light support yay ^^.

The light with it's color and the attenuation factor is simply a component that can be added to every entity to make it shine. Here is the example light of the player:

```
    com.Light(
        color=glm.vec3(0.6, 0.3, 1.2),
        attenuation=glm.vec3(0.1, 0.0, 1.0))
```

All lights that are bound to an entity with a `Transformation` component are collected in the `LightSetup` resource at the end of the frame. This setup is then loaded into the corresponding shaders. We do this every frame. This probably costs us some performance but enables us to use them more freely 😃

The shaders have a macro called `MAX_LIGHT_COUNT` this macro defines the maximum lights that can be passed into the shader and onto the next one. The current limit are 4 lights but we might increase it for the hand in. The fragment shader just goes through every light and passes the vector s (`to_light`) to the fragment shader. The fragment shader iterates trough all active lights (indicated by the uniform `light_count` variable). The lighting calculations are done for every light and the added to each other. The ambient light is added at the end one time.

This addition can result in a over saturation. Here is an example: We have a material with a `diffusion` of `(0.9, 0.1, 0.1)` and multiple infinite lights at the perfect positions and full color

```
color = diffusion * light_1 + diffusion * light_2 + ... + diffusion * light_10
// color => (9.0, 1.0, 1.0)
```

This is not a valid color but OpenGl simply excepts it as snow white. Having a usually red object being displayed as white might sound weird at first but is actually realistic. The same thing can be archived in the real world where we basically just see white when an object is hit by a lot of light

Limitations and possible improvements

- The current code always loads the entire light setup into the uniform variables even if they are not used effecting the currently drawn objects. This is done to avoid some extra code to deal with the selection of lights for the fragment shader. This limits the lights in the scene to the `MAX_LIGHT_COUNT` variable in the shader and costs us some performance.
- The current system supports specular lighting but no reflections them self. It might be worth to look into actual reflection rendering in the next project ;D