Ray3D Notes

Framebuffer Size

- Game.cpp → Init_Window()
- Framebuffer size could be different from the window's size

```
glfwGetFramebufferSize(window, &framebufferWidth, &framebufferHeight);
```

- This retrieves the size of the window's framebuffer - a collection of buffers *(e.g., color, depth, and stencil buffers)* used to render images to the screen.

```
glfwSetFramebufferSizeCallback(window, FramebufferResizeCallback);
```

- Setting a callback function that GLFW will call whenever the window’s framebuffer is resized. This helps in dynamically adjusting the viewport when the window size changes.

```
inline static void FramebufferResizeCallback(GLFWwindow* window, int fbw, int fbh)
{
    glViewport(0, 0, fbw, fbh);
}
```

- It updates the OpenGL viewport to match the new dimensions of the framebuffer.

GL_DEPTH_TEST: Depth testing is crucial for rendering 3D scenes correctly, ensuring that objects that are closer to the camera obscure objects behind them.

GL_CULL_FACE: Face culling allows OpenGL to skip rendering certain faces of objects based on their orientation.

glCullFace(GL_BACK): OpenGL culls the back faces *i.e.* the faces that are facing away from the polygon.

glFrontFace(GL_CCW): Ensuring that any polygons whose vertices are ordered counterclockwise will be considered the front face.

GL_BLEND: It helps in rendering transparent objects.

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA): *GL_SRC_ALPHA* means the source color is multiplied by its alpha value. *GL_ONE_MINUS_SRC_ALPHA* means destination color is multiplied by 1 minus the source's alpha.

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL): This will render polygons as filled shapes.

---

glm::lookAt()

```
//Syntax
glm::mat4 lookAt(const glm::vec3& eye, const glm::vec3& center,
    const glm::vec3& up);

//Code
viewMatrix = glm::lookAt(camPos, camPos + camFront, worldUp);
```

- ***eye:*** Position of the camera (camPos)

- ***center:*** point that the camera is looking at (camPos + camFront)
- ***up:*** up direction of the world (worldUp)

---

glVertexAttribPointer

- It maps your vertex buffer data to the vertex shader's attributes.

```
//Syntax
glVertexAttribPointer(GLuint index, GLint size, GLenum type,
    GLboolean normalized, GLsizei stride, const void * pointer);

//Example
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
    (GLvoid*)offsetof(Vertex,   position));

layout (location = 0) in vec3 v_position;  //Code from VertexShader.glsl
```

- ***index:*** Location of the vertex attribute in your shader program. Likein above example, the index would be 0, since *position* is bound to location 0.
- ***size:*** No. of components we are providing like in above example, since we are passing a *'vec3'*, we pass in size 3.
- ***type:*** Data type
- ***normalized:*** Normailzed to range [0, 1] for unsigned, and [-1, 1] for signed.
- ***stride:*** No. of bytes b/w consecutive vertex attributes.
- ***pointer:*** Offset in the buffer where the first component of the attribute data is located. It is passed in bytes, so it essentially a memory offset.

---

vs_position (Vertex Position)

- This is the **position** of each vertex in world space or view space.
- This helps to compute the distance and direction between a vertex and the light source, which influences lighting intensity.

vs_color (Vertex Color)

- This is the color of the vertex, often defined in the vertex shader.
- *Not implemented in my code but cn be used instead/along with textures.*

vs_texcoord (Texture Coordinates)

- These are the texture coordinates that tell the fragment shader how to map the texture image onto the 3D surface.
- *I have used this to fetch color data fromt he diffues & specular textures in the material.*

vs_normal (Vertex Normal)

- This is the normal vector at each vertex, which defines the direction the surface is facing.
- It's a crucial element for lighting calculations because it determines how light interacts with the surface.

Calculate Attenuation

```
float dist = length(light.position - vsPos);
return (light.constant / (1.0 + light.linear * dist + light.quadratic * pow(dist,2
```

- *light.constant* helps ensure that light doesn't completely disappear when objects are very close to the light source, *very close* to the light source, the attenuation effect is minor, and the light remains intense.
  - In simpler words, it is a constant "base level" of the light that doesn't diminish over distance.
- *light.linear* controls how much the light's intesnity decreases as the distance increases.
  - Higher value means light fades faster as you move away.
  - It's multiplied by *dist* to form a linear relationship.
- *light.quadratic* scales with square of distance.
  - It makes light decay more rapidly over longer distances. Higher the value, faster the light diminishes.
- ***Attenuation Formula:***

$$1.0/(constant + linear * dist + quadratic * dist^2)$$

  - So basically 1.0 *we* have used is just another constant.

Calculate Diffuse

```
float diffuse = max(dot(normal, lightDir), 0.0);

return (mat.diffuse * diffuse);
```

- *dot(normal, lightDir)*
  - Determines how much of the surface is facing the light.
  - Ranges from -1 to 1:
    - -1 = surface is facing away fromt the light. (no lighting)
    - 0 = surface is perpendicular to light. (no *direct* lighting)
    - 1 = surface is perfectly facing the light (max lighting)
- *max(..., 0.0)*
  - Just makes sure that the diffuse value stays positive.

Calculate Specular

```
vec3 reflectDir = reflect(-lightDir, normal);
float spec        = pow(max(dot(viewDir, reflectDir), 0.0), 48);
// You can change shininess (32, more popularly used) if needed

return (mat.specular * spec * texture(mat.specularTex, vs_texcoord).rgb);
```

- *reflect(-lightDir, normal)*
  - We negate lightDir because the function expects the incoming direction to be toward the surface.
- *spec*
  - Higher values create smaller, sharper, reflection of the light, and lower values the opposite.
