GPU Side
Create shader

BasicLight Vertex Shader

```
#version 330 core

// Vertex attributes
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;

// Uniforms
uniform mat4 uniform_mvp;

// the outgoing ocs normal to fragment shader
out vec3 normal_ocs;

void main(void) {
// pass the normal to the fragment shader
normal_ocs = normal;

// Required output
    gl_Position = uniform_mvp * vec4(position, 1.0);
}
```

BasicLight Fragment Shader

```
#version 330 core
layout(location = 0) out vec4 out_color;
uniform vec4 uniform_color;

// the uniform normal matrix for transforming the normals to eye space (ECS)
uniform mat4 uniform_normal_matrix;

// the incoming normal from the vertex shader
in vec3 normal_ocs;

void main(void)
{
// transform the normal to eye space (ECS)
vec4 normal_ecs = uniform_normal_matrix * vec4(normal_ocs, 0);
normal_ecs.xyz = normalize(normal_ecs.xyz);

// get the dot product between the normal and the view direction
// this simulates a light at the position of the camera
float light = max(0.0, dot(normal_ecs.xyz, vec3(0,0,-1)));
out_color = uniform_color * light;
}
```

CPU Side

Step 1. Variable declaration for shader and uniforms

```
ShaderGLSL* bl_glsl = nullptr;
GLint bl_program_id;
GLint bl_uniform_mvp;
GLint bl_uniform_normal_matrix;
GLint bl_uniform_color;
```

Step 2. Compile shader and create uniforms (in InitializeRenderer)

```
bl_glsl = new ShaderGLSL("BasicLighting");
shader_loaded = bl_glsl->LoadAndCompile();
if (!shader_loaded) return false;

bl_program_id = bl_glsl->GetProgram();

bl_uniform_mvp = glGetUniformLocation(bl_program_id, "uniform_mvp");
bl_uniform_normal_matrix = glGetUniformLocation(bl_program_id,
"uniform_normal_matrix");
bl_uniform_color = glGetUniformLocation(bl_program_id, "uniform_color");
```

Step 3. Draw

```
// Bind shader
glUseProgram(bl_program_id);
// Pass uniforms
glUniformMatrix4fv(bl_uniform_mvp, 1, false,
&model_view_projection_matrix[0][0]);
glUniform4f(bl_uniform_color, 1.0f, 0.0f, 0.0f, 1.0f);

// Create the normal matrix
glm::mat4x4 normal_matrix = glm::transpose(glm::inverse(world_to_camera_matrix
* object_to_world_matrix));

// Pass the normal matrix to the GPU as a uniform
glUniformMatrix4fv(bl_uniform_normal_matrix, 1, false, &normal_matrix[0][0]);

// Bind the sphere VAO and draw
glBindVertexArray(vao_sphere);
glDrawArrays(GL_TRIANGLES, 0, vao_sphere_indices);
```

Build VAO (changes are shown in red)

```
struct SphereVertexStruct
{
	glm::vec3 position;
	glm::vec3 normal;
};
void BuildSphereVAO2(float sphere_radius, int longitude_steps, int latitude_steps)
{
// Generate a vertex array object (VAO) to point to buffer objects
glGenVertexArrays(1, &vao_sphere);
// Set the VAO active
glBindVertexArray(vao_sphere);

// add vertex data
std::vector<SphereVertexStruct> sphere_data;

if (longitude_steps < 2) longitude_steps = 2;
if (latitude_steps < 4) latitude_steps = 4;

latitude_steps = 40;
longitude_steps = 20;

float phi_step = 2 * glm::pi<float>() / float(latitude_steps);
float theta_step = glm::pi<float>() / float(longitude_steps);
float phi = 0;
float theta = 0;
for (int j = 0; j < longitude_steps; j++)
{
	// temp variables to avoid calculating trigonometric values many times
	float costheta = glm::cos(theta);
	float sintheta = glm::sin(theta);
	float costheta_plus_step = glm::cos(theta + theta_step);
	float sintheta_plus_step = glm::sin(theta + theta_step);

	phi = 0;
	for (int i = 0; i < latitude_steps; i++)
	{
	// temp variables to avoid calculating trigonometric values many times
	float cosphi = glm::cos(phi);
	float sinphi = glm::sin(phi);
	float cosphi_plus_step = glm::cos(phi + phi_step);
```

```
        float sinphi_plus_step = glm::sin(phi + phi_step);

        SphereVertexStruct top_left;
        top_left.position.x = sphere_radius * sintheta * cosphi;
        top_left.position.z = sphere_radius * sintheta * sinphi;
        top_left.position.y = sphere_radius * costheta;
        top_left.normal     = glm::normalize(top_left.position);

        SphereVertexStruct top_right;
        top_right.position.x = sphere_radius * sintheta * cosphi_plus_step;
        top_right.position.z = sphere_radius * sintheta * sinphi_plus_step;
        top_right.position.y = sphere_radius * costheta;
        top_right.normal     = glm::normalize(top_right.position);

        SphereVertexStruct bottom_left;
        bottom_left.position.x = sphere_radius * sintheta_plus_step * cosphi;
        bottom_left.position.z = sphere_radius * sintheta_plus_step * sinphi;
        bottom_left.position.y = sphere_radius * costheta_plus_step;
        bottom_left.normal     = glm::normalize(bottom_left.position);

        SphereVertexStruct bottom_right;
        bottom_right.position.x = sphere_radius * sintheta_plus_step * cosphi_plus_step;
        bottom_right.position.z = sphere_radius * sintheta_plus_step * sinphi_plus_step;
        bottom_right.position.y = sphere_radius * costheta_plus_step;
        bottom_right.normal     = glm::normalize(bottom_right.position);

        sphere_data.push_back(bottom_left);
        sphere_data.push_back(bottom_right);
        sphere_data.push_back(top_right);
        sphere_data.push_back(top_right);
        sphere_data.push_back(top_left);
        sphere_data.push_back(bottom_left);

        phi += phi_step;
        }
        theta += theta_step;
}

GLsizei stride = sizeof(SphereVertexStruct);
int total_vertex_byte_size = sphere_data.size() * stride;
vao_sphere_indices = sphere_data.size();
```

```
glGenVertexArrays(1, &vao_sphere);
glBindVertexArray(vao_sphere);

GLuint vbo_sphere = 0;
glGenBuffers(1, &vbo_sphere);
glBindBuffer(GL_ARRAY_BUFFER, vbo_sphere);
glBufferData(GL_ARRAY_BUFFER, total_vertex_byte_size, &sphere_data[0], GL_STATIC_DRAW);

glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, stride, 0);
glVertexAttribPointer((GLuint)1, 3, GL_FLOAT, GL_FALSE, stride, (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

glBindVertexArray(0);
glError();
}
```