# COMP3811 – Exercise G.5 Lighting

## Contents



Exercise G.5 introduces a simple "$\mathbf{n} \cdot \ell$" lighting. You will implement a directional light with an ambient and diffuse contribution. To facilitate this, you will need to provide per-vertex normals to OpenGL. It will demonstrate how a simple binary format can be used to make mesh loading more efficient. Optionally, you will also be able to experiment with indexed meshes.

It is recommended that you have completed Exercises G.2-G.4 before this one. The practical components build on the G.4 solutions.

Note: You are allowed to reuse code from your solutions to exercises in the courseworks, *if and only if* you are the sole author of the code in question. Keep this in mind when working on the exercises. While you are encouraged to discuss problems with your colleagues, you should not directly share any code. If you find solutions to your problems online, study the solution and then implement it yourself. Never copy-paste code, and always make sure you understand the code that you write!

If you run into problems, you are expected to attempt resolving your issues yourself first, e.g., by debugging your program. Do not just immediately ask for help. Assistants may ask you to show/explain what you've attempted so far. Help will only be provided if you have made a fair attempt first.

*Recommendation: It might be tempting to just copy-paste the code from the PDF document into your solution while working through this exercise. I would recommend against this. You will gain far more by spending the small amount of extra time it takes to write the code yourself.*

## 1 Lighting

Make sure that you have your Exercise G.4 solution at hand. Transfer over select parts of the code:

- OpenGL Setup: perform the same setup as in G.4: enable the `FRAMEBUFFER_SRGB` state, enable depth testing and face culling, and set the clear color to e.g. a dark gray.
- Transformations: make sure you have valid world-to-camera space and projection matrices. You may want to reuse the arc-ball control setup.
- Shaders: You can reuse the shaders from G.4.

- Drawing: Draw something to ensure that things are set up appropriately.
- Implementations of the matrix functions and operators, specifically for `Mat44f`. Note that Exercise G.5 adds a few new functions to the `mat44.hpp` header: a `transpose` and an `invert` function.
- The creation code for the cylinder.
- Code for creating VAOs

Make sure the code runs, and is able to draw something. This will make debugging quite a bit easier down the road.

The `SimpleMeshData` structure defined in `simple_mesh.hpp` has been updated: a new `normals` member has been added. We will store the per-vertex normals here.

## 1.1 Normals (cylinder)

Generating normals for a cylinder is relatively straight forward. Like the cylinder, we can generate normals in groups: for the shell, and for each of the caps. The "base cylinder" in G.4 is aligned with the x-axis. The two caps lie in the $x = 0$ and $x = 1$ planes. Consequently, the normals on the caps point either in the $-x$ or the $+x$ direction. The shell is also relatively simple. The points are generated on circles around points on the x-axis. The normals are simply the vectors that connect the center point to the point on the shell. Therefore:

$$\mathbf{n} = \begin{cases} (0, y, z)\,, & \text{if point } (x, y, z) \text{ on shell} \\ (-1, 0, 0)\,, & \text{if point on cap at } x = 0 \\ (1, 0, 0)\,, & \text{if point on cap at } x = 1. \end{cases}$$

(You may want to verify that these normals are indeed unit length.)

We must also take care to transform the normals appropriately based on the passed in `aPreTransform` matrix. As outlined in the lectures, a normal matrix can be constructed from another transform $M$ by extracting the $3 \times 3$ submatrix of inverse-transpose of $M$:

```
Mat33f const N = mat44_to_mat33( transpose(invert(aPreTransform)) );
```

Exericse G.5 includes a `Mat33f` class (`vmlib/mat33.hpp`), and defines the `mat44_to_mat33`, `invert` and `transpose` methods for $4 \times 4$ matrices.

**Task:** Update the `make_cylinder` method to compute normals for each generated vertex. Ensure that the normals are transformed correctly and that they have unit length when returned in the `SimpleMeshData`.

Make sure to pre-compute `N` once in `make_cylinder` – do not recompute it for each normal. Computing matrix inverses, even when expressed as a straight-line program as is the case with the implementation in `vmlib/mat44.hpp`, is fairly expensive.

## 1.2 Drawing (debug)

The next step is to draw our cylinder. We will first just visualize the normals as colors, to verify that they are defined correctly.

**Task:** Update the `create_vao` function (`simple_mesh.cpp`) to define the normals as a third input vertex attribute in the VAO (in addition to the positions and colors). The normals have the same format as positions and colors in this example (three floats). The positions are vertex attribute zero, the colors vertex attribute one. We will therefore specify the normals as vertex attribute two. You will also need to create another VBO for the normals.

Next, we want to update the vertex shader to receive this newly defined input vertex attribute (index/location two). We will also want to transform the normals from model space to world space in the vertex shader, with a normal matrix.

**Task:** Update the vertex shader (`assets/default.vert`. It should receive an additional input:

```
layout( location = 2 ) in vec3 iNormal;
```

The normal matrix ($3 \times 3$ matrix) is passed to the vertex shader as a uniform:

```
layout( location = 1 ) uniform mat3 uNormalMatrix;
```

(Ensure that your code does not use uniform location one for something else!)

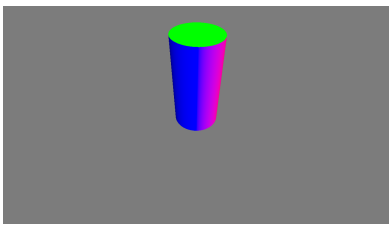

***Figure 1:*** *Debug drawing of a cylinder with normals. The normals are visualized as the colors. For example, the cap at the top is green = $(0, 1, 0)$ which corresponds to the normal pointing up along $+y$. The red and blue values on the cylinder's body can be interpreted similarly. Note that any value below zero will be clamped to zero.*

The vertex shader will have to pass the transformed normal to the rasterization stage, where it will be interpolated across the triangle and passed on to the fragment shader. Define an additional output:

```
out vec3 v2fNormal;
```

Finally, in the vertex shader's `main()`, transform the normal, and store the normalized result in the output:

```
v2fNormal = normalize(uNormalMatrix * iNormal);
```

**Task:** Update the fragment shader (`assets/default.frag`). It should receive the interpolated normal as an input:

```
in vec3 v2fNormal;
```

For now, we will renormalize the normal and then just copy it to the fragment shader's output. This allows us to visualize the normal. Implement this in the fragment shader's `main()` method:

```
vec3 normal = normalize(v2fNormal);
oColor = normal;
```

(Ensure that there are no other active assignments to `oColor` in the fragment shader.)

**Task:** Update the drawing code in the main loop (`main.cpp`) to compute the normal matrix and pass it to the shaders as a uniform matrix.

The normal matrix may be computed from the model-to-world transform that we have defined previously:

```
Mat33f normalMatrix = mat44_to_mat33( transpose(invert(model2world)) );
```

The $3 \times 3$ matrix is passed to the shader program with `glUniformMatrix3fv` (note the 3 instead of 4 in the function name):

```
// after glUseProgram( prog.programId() ) but before glDraw*():

glUniformMatrix3fv(
   1, // make sure this matches the location = N in the vertex shader!
   1, GL_TRUE, normalMatrix.v
);
```

**Task:** Create a cylinder and run the program. Compare to Figure 1. The image shows a single cylinder that is rotated to align with the $y$ axis:

```
auto testCylinder = make_cylinder( true, 128, {0.4f, 0.4f, 0.4f},
   make_rotation_z( 3.141592f / 2.f )
   * make_scaling( 8.f, 2.f, 2.f )
);
```

## 1.3 Shading

Exercise G.5 uses a simplified shading model based on (Blinn-)Phong shading. The simplified model includes only the ambient and diffuse terms. It uses a directional light (as opposed to a point light). Furthermore, it will use the vertex's color attribute ($v_c$) as both the ambient and diffuse reflectance ($k_a$ and $k_d$) material parameters.

The simplified model becomes

$$\begin{aligned} I_{\text{out}} &= I_a\, k_a + (\mathbf{n} \cdot \ell)_+\, I_d\, k_d \\ &= \left(I_a + (\mathbf{n} \cdot \ell)_+\, I_d\right)\, v_c \end{aligned}$$

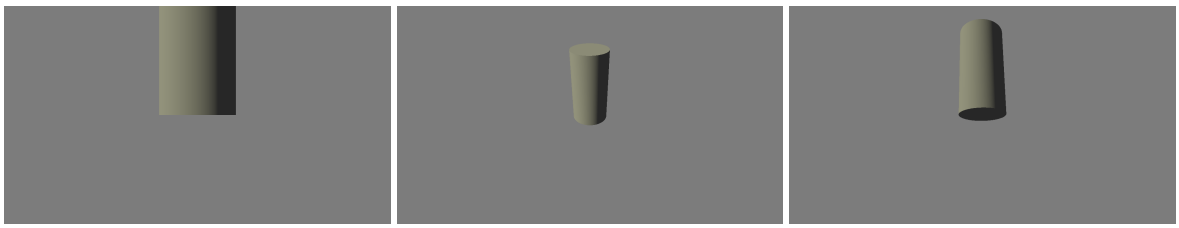

***Figure 2:*** *Cylinder lit with the simplified ambient-diffuse illumination model presented in Section 1.3. From left to right: default view at startup (looking straight at the cylinder's base), view from the top, and view from the bottom. The directional light is shining from the top-left, slightly behind the default camera. The top cap of the cylinder therefore receives illumination, whereas the bottom does not.*

Here, $(\mathbf{n} \cdot \ell)_+$ denotes a clamped dot product. The result of the dot product are clamped to the range $[0, 1]$. It can be implemented as $(\mathbf{n} \cdot \ell)_+ = \max(0, \mathbf{n} \cdot \ell)$.

To summarize, the model has the following parameters:

- Light direction $\ell$: a directional light source is defined by a single direction that is identical for all vertices (this is different from a point light, which has a position, and where the light direction is defined as the direction from the fragment position towards the light position). We will pass in the light direction as a uniform vec3 `uLightDir` in location 2.
- Diffuse illumination $I_d$. The diffuse illumination is passed in as a uniform vec3 `uLightDiffuse` in location 3.
- Ambient illumination $I_a$. We will use a global/scene ambient illumination that is also passed in as a uniform vec3 `uSceneAmbient`, in location 4.
- The material color (both $k_a$ and $k_d$) is defined as interpolated per-vertex color, which previous fragment shaders already received as a vec3 input attribute `v2fColor`.

**Task:** Implement the above in the fragment shader (`assets/default.frag`). First declare the additional uniform data:

```
layout( location = 2 ) uniform vec3 uLightDir; // should be normalized! ||uLightDir|| = 1
layout( location = 3 ) uniform vec3 uLightDiffuse;
layout( location = 4 ) uniform vec3 uSceneAmbient;
```

Then implement the simplified lighting computations (using the `normal` computed in the previous section):

```
float nDotL = max( 0.0, dot( normal, uLightDir ) );

oColor = (uSceneAmbient + nDotL * uLightDiffuse) * v2fColor;
```

**Task:** Update the drawing code in the main loop (`main.cpp`) to set the newly defined uniform values:

```
Vec3f lightDir = normalize( Vec3f{ -1.f, 1.f, 0.5f } );
glUniform3fv( 2, 1, &lightDir.x );

glUniform3f( 3, 0.9f, 0.9f, 0.6f );
glUniform3f( 4, 0.05f, 0.05f, 0.05f );
```

Note that the locations (first argument to each `glUniform*` call) must match the locations declared in the fragment shader. We are assuming that the light direction in the uniform variable is normalized. (Remember that you must set uniform values after binding the program, but before the relevant draw calls.)

Verify that you get the expected output (Figure 2). The diffuse illumination is slightly yellow (RGB = $0.9, 0.9, 0.6$), but the ambient one is a neutral (but very weak) grey (RGB = $0.05, 0.05, 0.05$). The light is coming from the top-left, and from slightly behind the default camera position.

The lighting and the normals should "stick" to the cylinder. As the camera moves around, the lighting of the cylinder should not change (albeit you will be looking at the cylinder from different directions, which means that you will see different sides with different illuminations.).

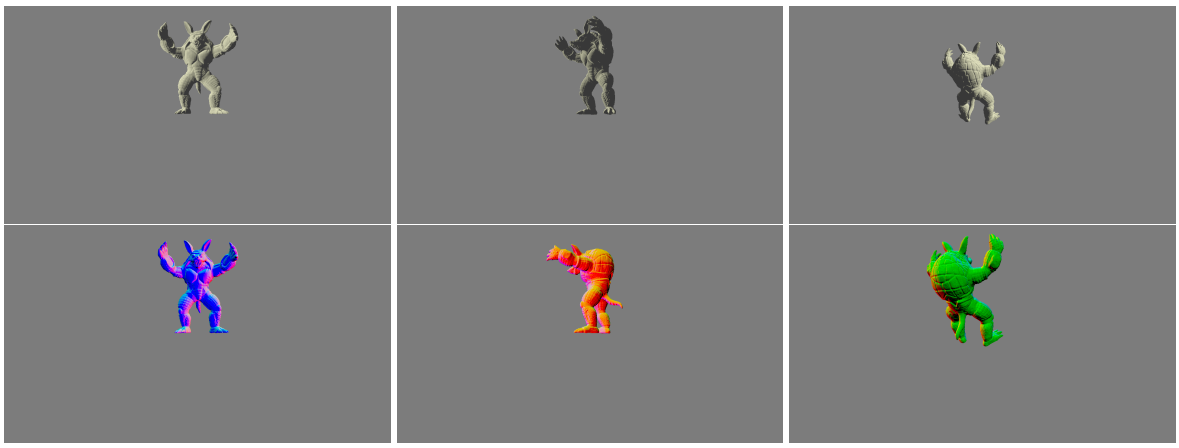

***Figure 3:*** *Top row: shaded Armadillo models, with the directional light defined in Section 1.3. Bottom row: debug renderings of the Armadillo model, visualizing normals as colors. Since the model is fixed relative to world space and shading is performed in world space, the normals should "stick" to the model, i.e., not swim across the model as the camera moves around.*

### 1.4 Normals (Armadillo)

You can use the same shader and setup to draw the Armadillo mesh. However, we will need to provide per-vertex normals for the mesh as well. There are a few different ways of getting normals. Exercise G.5 will use a custom binary file format that contains the relevant data, to demonstrate how such can be utilized easily and efficiently.

Exercise G.5's code includes the Armadillo mesh in the custom file format. Find it in the `assets` folder, as `assets/Armadillo.comp3811bin`. The file format is outlined in the `readme.md`. It includes per-vertex positions, colors and normals, and is stored as an indexed mesh. The file is around 10MB. This is less than the original Wavefront OBJ file, which weighs in at about 28MB. Recall that the Wavefront OBJ file does not store per-vertex colors either. Furthermore, the binary format can be read very easily and efficiently (in about six calls to `std::fread` or similar).

> The file format is kept simple. It omits certain guards and checks that can be useful. For example, it does not contain (easily verifiable) information on the expected endianness of the data. The loader makes no attempt to check for this. Therefore, attempting to load the included file (created on an x86 little-endian machine) on a big-endian machine will likely fail in strange ways. Fortunately, big-endian machines are quite rare these days. Other endians are even less common.

For your convenience, a loading function is provided. It will load the data from the binary format, and then convert into a `SimpleMeshData` structure (which includes turning the indexed mesh into a triangle soup; this is nevertheless just a few additional lines of code). All in all, it is by far simpler than even a basic Wavefront OBJ loader, let alone an optimized library such as *rapidobj*.

**Task:** Briefly study the `load_simple_binary_mesh` function declared in `loadcustom.hpp` and defined in `loadcustom.cpp`.

**Task:** Use the `load_simple_binary_mesh` function to load the provided Armadillo mesh with normals and render it the simple lighting model implemented previously. Visualize the normals to verify that they are being loaded correctly. Compare your results to Figure 3.

## 2 (Optional) Indexed mesh

*Note: Indexed mesh rendering is not required for the 2023/24 Coursework. If you are pressed for time, you can safely skip this part.*

The simple binary format stores the Armadillo mesh as an indexed mesh. In the previous section, it is converted into a triangle soup before rendering. This keeps the pipeline simpler, but is less efficient for rendering. In fact, the indexed mesh in the file format has been optimized for rendering, with indices and vertices being reordered to aid the efficacy of caches, such as the post-transform cache and standard memory caches.

To render an indexed mesh, you will need to load the (unchanged) per-vertex data into separate VBOs. Additionally, you will need to load the index data into an *index buffer object* (IBO). In OpenGL, the IBO is a buffer

object of type `GL_ELEMENT_ARRAY_BUFFER`. The IBO must be associated with VAO, which is done by binding it (`glBindBuffer`) as `GL_ELEMENT_ARRAY_BUFFER` *while* the corresponding VAO is bound.

The following will outline two possible options. The first option modifies the existing code, meaning that it will require less new code. The second option takes an alternative approach, which is slightly more efficient, but may require a bit more new code. The drawing code will be identical, regardless of the option chosen.

## Option A: From existing code

In this first option, you will start with the `SimpleMeshData` structure. Add a `std::vector<std::uint32_t>` buffer, which will be used to store the indices. Write a `load_simple_binary_mesh`-analogue function that loads the relevant data. Make sure that you load the data in the right order. (Do not convert the indexed data into a triangle soup.)

Then modify the `create_vao` function to additionally create an IBO, upload the index buffer data into it, and bind it into the VAO state. The number of vertices to be drawn is equal to the number of indices.

## Option B: Loading directly into BOs

The second option aims to skip the intermediate step of allocating CPU buffers and storing the data there first. Instead, we will create the VBO and IBO buffer objects, and ask OpenGL to give us a pointer to the memory in those buffers. We will then read the file's contents directly into this memory.

For example, to load the indices (assume this takes place after having read the file magic and the number of vertices and indices):

```
// Read IBO
GLuint ibo = 0;
glGenBuffers( 1, &ibo );

glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, ibo );
glBufferData( GL_ELEMENT_ARRAY_BUFFER, sizeof(uint32_t)*numIndices, nullptr, ▽
  ▷ GL_STATIC_DRAW );

void* iboptr = glMapBuffer( GL_ELEMENT_ARRAY_BUFFER, GL_WRITE_ONLY );
fread_( iboptr, sizeof(uint32_t)*numIndices, fin );
glUnmapBuffer( GL_ELEMENT_ARRAY_BUFFER );
```

Here, we create the IBO (type `GL_ELEMENT_ARRAY_BUFFER`). We the allocate the storage with `glBufferData`, passing `nullptr` as the 3rd argument to indicate that the memory should be left uninitialized. Then, we use `glMapBuffer` to ask OpenGL for a pointer to the buffer's memory. We pass this pointer to `std::fread` to read the file's contents directly into that memory. Finally, we unmap the buffer `glUnmapBuffer` again. The last step ensures that the new data is synchronized to e.g. GPU memory, if it was not possible to get a direct pointer there or to force caches on the way to be flushed.

Per-vertex attribute data would be handled similarity, albeit with standard VBOs instead of an IBO. Assembling the VBOs and IBOs into a VAO is largely unchanged from previous attempts.

The reference implementation for this uses a function

```
GLuint load_indexed_binary_mesh( char const* aPath, std::size_t& aVertsOut );
```

as a single-step loading function. (You will need to declare and define this function if you opt for this solution.)

## Drawing

Drawing with indices is only superficially different from drawing a triangle soup. Instead of `glDrawArrays`, we now use `glDrawElements`. The function takes four arguments. The first one is the draw mode, which is identical to `glDrawArrays`. For drawing triangles, it is set to `GL_TRIANGLES`. The second argument specifies the number of elements to draw with. Each index refers to one element, so we set this to the number of indices to draw the whole mesh. The third argument specifies the type of the indices, which in this case is `GL_UNSIGNED_INT` (32-bit unsigned integers). Finally, we can specify an offset, which defines where in the IBO the first index occurs at. We set it to `nullptr` to start with the first index defined in the IBO:

```
glBindVertexArray( vao );

// Set up other stuff, e.g., per-instance uniform values

glDrawElements( GL_TRIANGLES, count, GL_UNSIGNED_INT, nullptr );
```

## Performance

You may want to compare performance when drawing the Armadillo mesh as a triangle soup compared to an indexed mesh. The differences are most visible when drawing many instances (though the exact number required will depend on the hardware that you are running this on).

Note that measuring performance on the CPU will not necessarily give you a good result. Most of the time will be spent on the GPU. To measure GPU time, you can use OpenGL's query objects. I recommend using the `GL_TIMESTAMP` query. The general strategy is to create a pair of query objects (`glGenQueries`) during setup. The `glQueryCounter` method can be used to record a GPU-based time stamp just before drawing and just after drawing (hence the need for a pair of query objects). The results are finally retrieved with `glGetQueryObjectui64v`.

Note that retrieving the query results will block until the results are available, introducing a synchronization point between GPU and CPU. One typically wants to avoid these (which is possible by using multiple pairs of queries, and only retrieving the results several frames later, when they are guaranteed to have completed). However, in this case, the extra complexity is probably unnecessary.

Figure 4 shows another 400 Armadillos arranged in a $20 \times 20$ grid. Using an indexed mesh approximately halves the rendering time in a quick informal test. This can likely be optimized further.

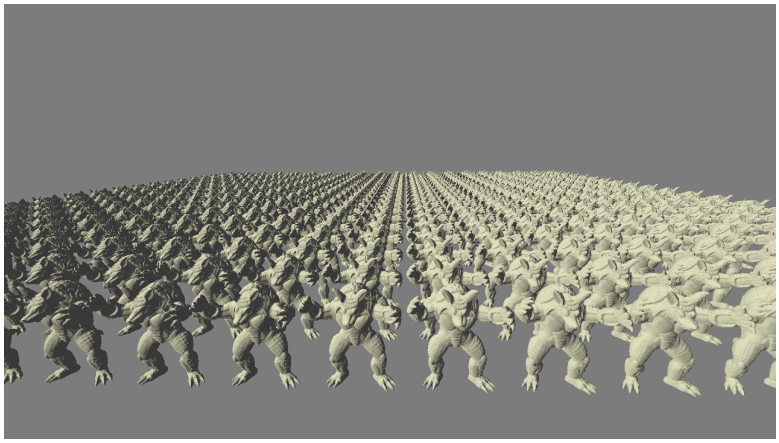

***Figure 4:*** *According to not very thoroughly verified internet resources, a group of armadillos is apparently referred to as a "roll of armadillos". Hence, this would be a lit roll of armadillos. On a slightly more relevant note: 400 Armadillos take around 30ms when rendered as a triangle soup, and about half (∼15ms) when rendered as indexed meshes (measured on a NVIDIA RTX 3070 Ti).*