# Computer Graphics

*Exercises 5*

1) Currently, the Vertex contains attributes that were chosen for the sake of simplicity and are in chronological order of implementation. See the **left** picture for the current Vertex and the **right** for a more optimized version. Replace your current Vertex struct with the new one and adjust everything such that the renderer works as it did before. Keep in mind that the position needs to be vec4 with **w=1.0** when multiplied with a matrix! Shader syntax that may be useful:

```
vec3 a = vec3(1.0, 1.0, 1.0);
vec4 b = vec4(a, 1.0);
vec3 c = b.xyz;
vec3 d = b.rgb;
```

```
struct Vertex {
    glm::vec4 position;
    glm::vec4 color;
    glm::vec2 uv;
    glm::vec3 normal;
};
```

```
struct Vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec4 color;
    glm::vec2 uv;
};
```

2) Having multiple separate render pipelines may often be the case, but it should be reduced to the minimum number possible. In our case, we have two separate pipelines for meshes with **vertex colors** and **texture colors**. The goal of these mini-tasks is to merge both render pipelines into a single one that can handle both types of meshes.
   a) Create a material.hpp file containing a **Material** struct that holds the following:
      1) specular: a float that controls the total specular light
      2) specular_shininess: another float that controls the surface shininess
      3) texture_contribution: the final float that interpolates between **vertex colors** and **texture colors** with a value range from 0 to 1.
   b) Pass both the **uv coordinates** and **vertex colors** from the vertex to fragment shader. Use the **mix** function to mix the vertex and texture colors together, using the texture_contribution as the interpolation value like so:
   vec4 color = mix(color_vert, color_tex, texture_contribution);
   c) Create a **Material** for each of the cubes, one of which should have texture_contribution of 0, the other of 1. Render this using only a single pipeline. The Material struct will need a bind() function to use in shaders.
3) Create a **model.hpp** file in the *include/entities/* folder containing the **Model** struct. This should be the main object for holding all the info for renderable objects, such as **Transform**, **Mesh**, **Texture**, **Material**. Give it the standard functions that are present on similar structures:
   void init() {} // initialize without texture
   void init(const char* tex_path) {} // initalize with texture
   void destroy() {} // destroy all the members
   void draw() {} // bind all the members and then draw the mesh

4) Currently, we only have a **cube** mesh with no other options within the **Mesh** struct. Another frequently used primitive shape is a **sphere**. There are many ways to create one with different types of properties; see https://www.songho.ca/opengl/gl_sphere.html for some available options. Pick one and implement it for our mesh. Don't worry about uv-coordinates, as we do not have a texture for spheres.
   Since we want to keep the cube, create some sort of option that allows a choice between which mesh should be created (parameter in init?).