# AC41001/AC51008 Graphics

## Lab Sheet 3: Lighting

# Aims

* Run the **lab3start** example program.
* Write GLSL vertex shader code for directional lighting
  + Ambient, diffuse, specular and emissive.
* Change the light source direction

### Setting up the project

For this week’s lab, you may choose to extend last week’s lab or use **lab3start**. Only use your own code if you got as far as defining a perspective projection.

Download the **lab3start** zip example and the associated **lighting\_Example\_objects** zip in the examples directory in the lighting topic on Blackboard. Set up a project in your preferred setup (e.g. Visual Studio 2013) and copy/import the source files into the new project. Build and run it, you should see a cube and a sphere. Keyboard controls have been added for rotation, translation and scaling. Use the keyboard controls to start move the objects around.

### 2. Ambient light

In your vertex shader, define an ambient light colour, and set the output colour to be ambient colour, e.g.

vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);

fcolour = ambient;

Try making the ambient light a proportion of the diffuse colour instead: e.g.

vec4 ambient = diffuse\_colour \* 0.2;

See what the effect is?

### 3. Diffuse light

There are quite a few steps required to calculate diffuse lighting so to help I’ve given some detailed steps to define lighting below. Feel free to implement your own version if you prefer (and are confident!). We also need to be careful about working with 3D vertex positions and homogeneous coordinates so don’t write it all at once, test each stage as you implement it. The input vertices and normal are in 3D space, the transformations are in homogeneous cords. It is easy enough to switch between the two. For example:

vec3 x3 = x4.xyz / x4.w;

vec4 x4 = vec4(x3, 1.0);

1. Define a light direction (e.g. a vec3).
2. Create a model\_view matrix by multiplying the model and view together. The order is important, it should be model\_view = view \* model;
3. Define the vertex position in homogeneous cords (convert to a vec4)
4. Define the mat3 normal model\_view transformation matrix using the inverse transpose function as discussed in the lecture slides. Note that if we don’t scale, we can just convert the mat4 model-view matrix (from step 2) to a mat3 but the inverse transpose matrix is better, e.g.

mat3 normal\_matrix = mat3(mv\_matrix); // or

mat3 n\_matrix = transpose(inverse(mat3(mv\_matrix)));

1. Transform the normal by the normal matrix (multiply). Note that a better way to do this may be to calculate the normal matrix in your application and define it as a uniform. See the slides from Lecture 8. The order of operands is important again (vec = matrix \* vec);
2. Normalise both the light direction and the transformed normal– use the function normalize(). E.g. light\_dir = normalize(light\_dir);
3. Calculate the diffuse component as the dot product between the normalised normal and light direction. Use the function dot(). (Note you should also limit it to a max of 0 using the max() function).
4. Calculate the diffuse lighting of the vertex by multiplying the diffuse component and the diffuse\_colour.
5. Set fcolour (the output colour) to be the ambient plus the diffuse values.

Save and run, you should see the objects with diffuse lighting implemented. Note: you should refer to the slides on lighting for help (Lectures 7 and 8).

Try combining your ambient and vertex colours (adding or multiplying them together, which is best?), see what that does? You could also define your ambient colour in your C++ code and pass it into your vertex shader as a **uniform variable**.

### 4. Add a specular component (Phong)

You could try to implement Phong specular or Blinn-Phong (see the lighting slides) for the theory. Blinn-Phong is more efficient but you need to calculate the half-vector from the camera position and light position so although less efficient, Phong specular can be a simpler calculation because you can use the reflect() GLSL function to easily calculate the reflected beam. Here is the general principle, you will need to adapt it to work with your code.

1. Define the vertex position transformed by the model-view transformation.
2. Define the shininess component by setting the value of a float variable, e.g. 8.0;
3. The specular reflection calculation is tricky so I’ve given it to you below with comments for further explanation. Note you may need to modify this to fit in with your code (e.g. variable names).

// V is the unit length direction vector of the vertex position to the origin

// P is the vertex position transformed by the model-view matrix.

// R is the reflected light beam of the plane defined by the vertex normal

// Specular colour is an RGB colour (e.g. white)

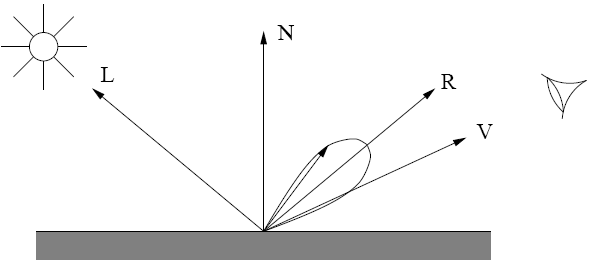
vec3 V = normalize(-P.xyz);

vec3 R = reflect(-L, N);

vec3 specular = pow(max(dot(R, V), 0.0), shininess) \* specular\_colour;

1. Set fcolour to be the sum of the ambient, diffuse and specular components.

Note that you should refer to the slides on specular lighting.



### 5. Add an emissive component

Add an optional emissive component.

Note that you should limit your colour values to the range of 0 to 1. See GLSL functions max, min or clamp.

### 6. Change the light source direction.

Define a light source position in your application code. Pass this to the shader as a uniform variable and use that instead of the previously hard-coded value. Define user controls to modify the light source position in your application.

### 7. Switch between lighting modes

Define a variable to control the lighting mode in your application (e.g. GLuint lightmode). Add user controls to modify it, pass it to your vertex shader and use to control whether ambient, diffuse, specular and emissive lighting is enabled (e.g. cycle through lighting modes).

### 8. Further exercises (optional)

* Implement Blinn-Phong specular reflection instead or (or as well as) Phong specular. (*hint: see lecture slide on the Blinn-Phong, half-vector, specular calculation)*.
* Look through the source code, look at how the sphere is constructed, try changing the number of latitudes and longitudes that define the sphere to make a higher or lower resolution object.
* Modify the colour of your light source and use that colour to combine with the vertex colours.
* Create a second light source in a different position, e.g. a red-light source and a white light source.
* Experiment with moving the view source position. Translate the view in the application (e.g. with rotations), then translate the light source position also with the view matrix in your application before passing it to the vertex shader. Run the program to see if the effect is as you think it should be.