Lab 4: Shading, Lighting and Shaders

Introduction

In this lab we'll look mostly at things we encountered in Lecture #5. The tasks aim to give an idea of how to calculate lighting using shaders in OpenGL. We thus have two main goals, understand how to calculate lighting, and understand how shaders work in OpenGL.

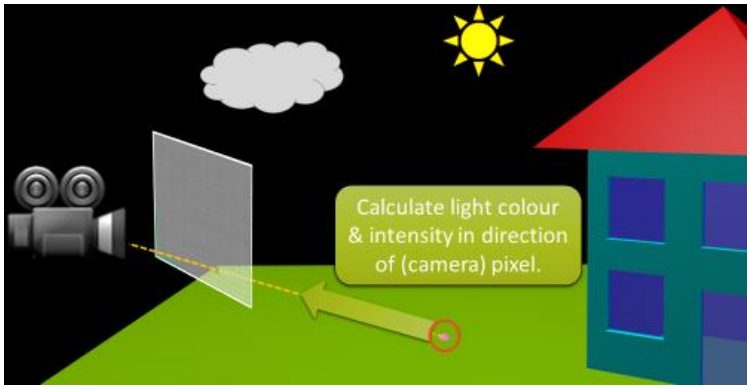
General instructions.

As before, *future* lab/lecture material in a module called 'magic'. You are free, encouraged even, to look at this code, which will be commented to explain what it is used for and how it works, but it will be explained in the future (or it does not need explanation for this course). Also, there is a module called 'lab_utils' where code that we have already covered is placed – remember that this is there for you to look at to recall how things are done. The labs are designed to tie in with the lectures, but are much more concrete and specific to OpenGL, whereas the lectures aim at a more general level.

When you have finished the tasks, or if you feel uncertain about anything, please speak to a tutor. This is not required to pass, but is a good idea to make sure you have not missed anything important.

Part 1 – Lighting

When talking about shading and lighting remember that the goal of it all is computing the colour for a single pixel. The pixel represents a point-sample of the scene, and for real-time purposes this is found by rasterizing the triangle. The pixel centre thus has a corresponding position in world or view (or model) space, which need to be used in the lighting calculations.



1. The basic shader setup

Before we get stuck into the lighting computations we'll set up the basic shaders that we will work in. In OpenGL a shader is a (usually short) program that is associated with some specific *programmable* stage of the GPU hardware pipeline. Shaders are *not* written in python, but in a language called GLSL which is part of the OpenGL specification[1]. The source code is uploaded to the OpenGL driver as a text string, which then compiles the code to machine code for the GPU. The language is a dialect of 'C' with a number of extensions and built-in functions.

Vertex Shader:

We have briefly met with a vertex shader in an earlier lab. The vertex shader is executed once for each vertex in the mesh being rendered its main job is to perform transformations on the vertex

[1] https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.3.30.pdf
The OpenGL man pages include GLSL functions: https://www.khronos.org/registry/OpenGL-Refpages/gl4/

data for later stages. The one thing it must do is provide *clip-space* coordinate of the vertex by writing to '`gl_Position`', which gets passed the rasterizer, which in turn samples screen-space triangles into fragments. To compute lighting we need two critical pieces of information:

1. The *position* of the sample we're computing lighting for.
2. The *normal* of the surface at this position.

The position is required to be able to compute the direction to the light, and the normal is required as it is critical in determining the intensity of the incoming light. Let's write a vertex shader:

```
#version 330
// input attribute variables
in vec3 positionAttribute;
in vec3 normalAttribute;
in vec2 texCoordAttribute;

// uniform variables
uniform mat4 modelToClipTransform;
uniform mat4 modelToViewTransform;
uniform mat3 modelToViewNormalTransform;

// output variable block
out VertexData
{
  vec3 v2f_viewSpaceNormal;
  vec3 v2f_viewSpacePosition;
  vec2 v2f_texCoord;
};

// main function (program entry point, unlike python!)
void main()
{
  gl_Position = modelToClipTransform * vec4(positionAttribute, 1.0);
  v2f_viewSpaceNormal = normalize(modelToViewNormalTransform * normalAttribute);
  v2f_viewSpacePosition =
      (modelToViewTransform * vec4(positionAttribute, 1.0)).xyz;
  v2f_texCoord = texCoordAttribute;
}
```

The variables declared at the top prefixed with '**`in`**' are input *vertex attributes*. Remember that this program is executed once for each vertex, e.g., three times for a triangle, and each time the '**`in`**' variables have a different value. In the lab program the actual data is loaded by the ObjModel class from a file with '.obj' extension and uploaded to OpenGL. The ObjModel looks for the names '`positionAttribute`' etc in the shader, and ensures they are bound to the correct data array. You can see how the data looks in the '.obj' files as it is a text based file format. Note that the variables also have a type declaration, which tells the compiler (and the programmer) what data type the variable represents – this is in contrast to Python which is *dynamically* typed (meaning that the types of variables may change at run-time), there are a number of [built-in types](https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)) in GLSL[3] that represent the most common linear algebra types that we need to do graphics.

The next three variables are declared with the prefix '**`uniform`**'[4], which means that the variable is the same for each of the executions of the program. We set these variables from our python code

[2] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/normalize.xhtml
[3] https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)
[4] https://www.khronos.org/opengl/wiki/Uniform_(GLSL)

prior to drawing any geometry. Here, they are used to represent transformation matrices that we need to transform the vertex data from model space to other spaces.

Next we have an 'output block' (**out** `VertexData`) which is a way to group several output variables into an *interface block*. This makes OpenGL validate that the fragment shader has an identical '**in**' interface block, which helps reduce bugs. The '**out**' keyword means variables that the vertex shader program will write to and which will be passed along to the rasterizer to be interpolated over the triangle before being passed to the fragment shader as input[5].

Lastly, we have the '`main`' function. This function is the program entry point and is called by the system (unlike Python which starts at the top of the program). The first line takes care of outputting a clip-space position to OpenGL:

```
gl_Position = modelToClipTransform * vec4(positionAttribute, 1.0);
```

The '`positionAttribute`', which is in model space is first extended to a 4D homogeneous point by appending a 1, and is then transformed by the '`modelToClipTransform`' transformation matrix. The built-in output variable '`gl_Position`[6]' is 4D vector (**vec4**). The hardware will take whatever is written to this variable and transform that to screen-space once the shader is finished. The remaining three lines transform vertex data to other spaces and writes to the user-defined '**out**' variables. Note that we transform both position and normal to **view space**, as we intend to compute shading in this space. View space is a sensible space to work in, since we often need to use the direction to the eye which is at the origin. Clip space is 4D and under projection, so is not suitable. World space is also fine, and does make some things such as reflections easier.

The source code for the vertex shader is placed in the file 'vertexShader.glsl'. We have put it in a separate file, rather than a string in the python program, to make it possible to re-load the shader while the program is running. This will make it much easier to experiment with the shader code. The lab program automatically re-loads the shaders every second or so.

```
def update(dt, keys, mouseDelta):
  g_reloadTimeout -= dt
  if g_reloadTimeout <= 0.0:
        reLoadShader();
        g_reloadTimeout = 1.0
```

If the compilation fails it prints the error messages to the consoles and keeps the previous shader, so keep an eye on the program output.

Fragment Shader

The next programmable stage is the fragment shader. It is executed once for each fragment – recall that a fragment is a pixel-aligned sample of the triangle. The goal of the fragment shader is to compute the colour of the fragment, which can then be merged into the framebuffer at the pixel position it represents. A fragment shader ***has no possible way*** to affect ***where*** in the framebuffer the colour is written to, this is determined by the geometry stages and the rasterizer.

```
#version 330
// Interpolated input from the vertex shader
in VertexData
{
  vec3 v2f_viewSpaceNormal;
  vec3 v2f_viewSpacePosition;
```

[5] https://www.khronos.org/opengl/wiki/Vertex_Shader#Outputs
[6] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/gl_Position.xhtml

```
  vec2 v2f_texCoord;
};

// Material properties set by ObjModel.
uniform vec3 material_diffuse_color;
uniform float material_alpha; // Note: represents opacity
uniform vec3 material_specular_color;
uniform vec3 material_emissive_color;
uniform float material_specular_exponent;

// Textures set by ObjModel
uniform sampler2D[7] diffuse_texture;
uniform sampler2D specular_texture;
// Other uniforms used by the shader
uniform vec3 viewSpaceLightPosition;
uniform vec3 lightColourAndIntensity;
uniform vec3 ambientLightColourAndIntensity;

out vec4 fragmentColor;

void main()
{
  vec3 materialDiffuse = texture[8](diffuse_texture, v2f_texCoord).xyz
                         * material_diffuse_color;

  fragmentColor = vec4(materialDiffuse, material_alpha);
}
```

First up, we have the input data block, which must match that of the vertex shader. These variables contain the interpolated attributes from the vertices for the sample position the fragment represents (e.g., for a sample in the centre of the triangle the attributes will be the average of the three vertex values).

Second, we have a large group of **uniform**s. The first two groups are material parameters that are set by the ObjModel, that define the properties of the material such as reflectiveness and opacity. The data in these are loaded from the '.mtl' file referenced by the '.obj' file – again, this is a text based format so you can easily inspect and modify it. You can also inspect and manipulate the values using the UI.

The last group of **uniform**s contain information about the light in the scene, and are set by the program. For example (in 'lab4_template1.py'), to copy the value of the python variable '`g_lightColourAndIntensity`' to the GLSL uniform variable named '`lightColourAndIntensity`' in the previously compiled and linked shader program represented by the '`g_shader`' variable:

```
…
lu.setUniform(g_shader, "lightColourAndIntensity", g_lightColourAndIntensity)
…
```

The function '`setUniform`' in 'lab_utils.py' follows the pattern:

```
def setUniform(shaderProgram, uniformName, value):
  loc = glGetUniformLocation[9](shaderProgram, uniformName)
  if isinstance(value, float):
    glUniform1f(loc, value)
```

[7] https://www.khronos.org/opengl/wiki/Sampler_(GLSL)
[8] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/texture.xhtml
[9] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glGetUniformLocation.xhtml

```
…
  elif isinstance(value, (np.ndarray, list)):
…
    if len(value) == 3:
      glUniform3fv(loc, 1, value)
```

It just checks the type of the argument to call the correct OpenGL function to set the value. The call to '`lu.setUniform`' above thus is equivalent to:

```
…
loc = glGetUniformLocation(g_shader, "lightColourAndIntensity")
glUniform3fv[10](loc, 1, g_lightColourAndIntensity)
…
```

Next there is an '**out**' variable[11]. The last (as in before the main function returns) value written to this variable is the colour that will be merged into the framebuffer. Recall that the fourth component 'w' or 'a' is often used to represent the opacity, if blending is enabled. If blending is not enabled this is just copied to the pixel. Typically there will only be one output variable from a fragment shader. However, for more advanced effects, or to implement *deferred shading*, it is common to use Multiple Render Targets (MRT), as these require storing more data for each pixel.

Finally, there is the '`main`' function, just as for the vertex shader. This function is called once for each fragment produced when drawing the geometry. The implementation shown here first samples the texture '`diffuse_texture`' and multiplies the returned colour value by the '`material_diffuse_color`'. If the material does not have a texture, ObjModel provides a default texture that is just a single white pixel (i.e,: $(1,1,1)$). This combined value represents the diffuse reflectance of the material. In this basic setup, we just return this as the fragment colour. This is the shader we will spend this lab working most of the time so make sure you understand how it works before proceeding.

2. Incoming direct light

The first thing we need to calculate is the amount of light arriving at the point we're shading. Obviously, if no light arrived, then it cannot be reflected towards the eye. The incoming light intensity is proportional to the cosine of the angle between the surface normal and the direction **to** the light.

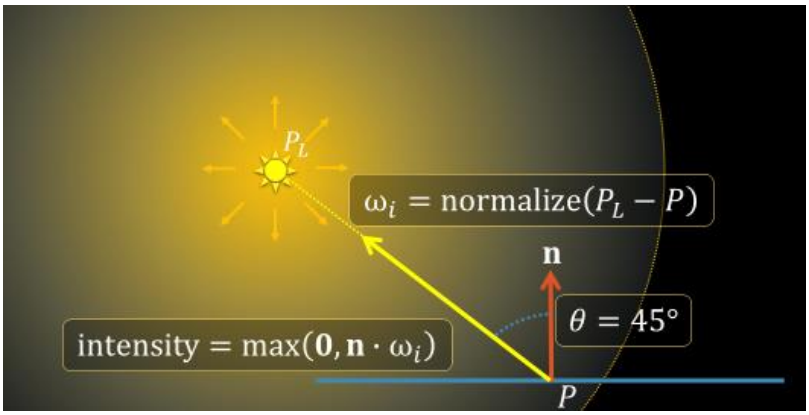


Figure 1. Incoming light intensity is proportional to the cosine of the angle between the surface normal and the direction to the light.

So the first thing we need to calculate is the direction towards the source of the light. The light position (in view space) is available to the fragment shader (and set by the program) as '`viewSpaceLightPosition`'. The current point is represented in the variable '`v2f_viewSpacePosition`', which is an interpolated variable passed from the vertex shader.

Add the following to the main function of the fragment shader to compute the normalized direction towards the light from the shading point in view space.

[10] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glUniform.xhtml
[11] https://www.khronos.org/opengl/wiki/Fragment_Shader#Outputs

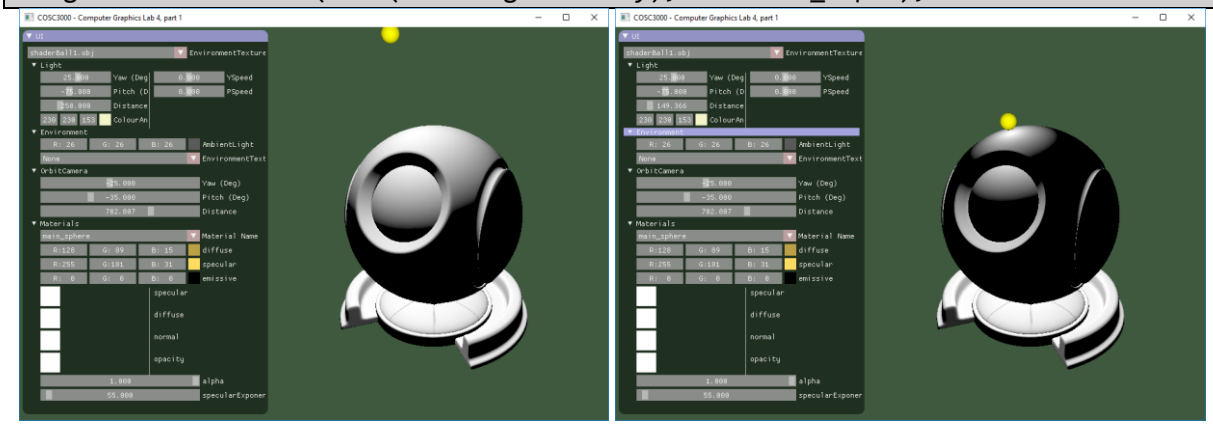
```
vec3 viewSpaceDirToLight =
                normalize(viewSpaceLightPosition- v2f_viewSpacePosition);
```

Now using this it is a simple matter to compute the incoming light intensity as:

```
vec3 viewSpaceNormal = normalize(v2f_viewSpaceNormal);
float incomingIntensity = max(0.0, dot(viewSpaceNormal, viewSpaceDirToLight));
```

The seemingly redundant normalization of the interpolated normal is useful as the linear interpolation does not maintain the unit-length property of the normal. To see the result visually, and this is also the primary way to debug shaders available to us, output the incoming intensity as the colour of the pixel:

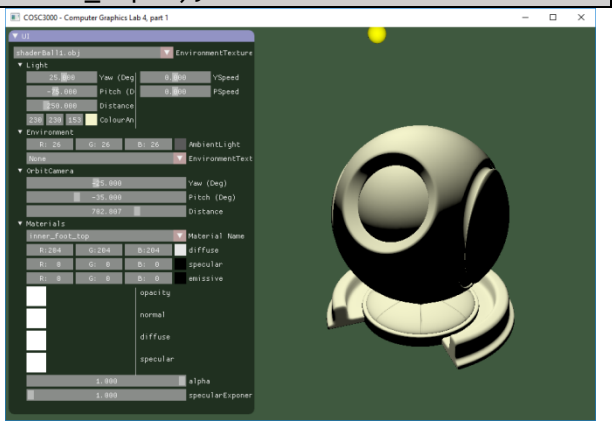
```
fragmentColor = vec4(vec3(incomingIntensity), material_alpha);
```



We see that surfaces oriented towards the light are affected by the light. Moving the light closer means more of the top sphere is facing away from the light. This quantity actually represents the *proportion* of the incoming light that arrives at the surface. It must be used to modify the light that is emitted by the light source to have the correct colour and maximum intensity (we are ignoring fall-off due to distance at the moment). Let's change to the below:

```
vec3 incommingLight = incomingIntensity * lightColourAndIntensity;
fragmentColor = vec4(incommingLight, material_alpha);
```

As you can see the light is now tinted slightly yellow. You can play around with the intensity and colour in the interface.



So this gives us the first step in the shader, now we need to calculate how much is reflected.

3. Diffuse Lambertian Reflection

Recall from Lecture #5 that the Bidirectional Reflection Distribution Function (BRDF) for lambertian diffuse is *constant*. Thus, we just have to multiply the incoming light with a constant, and voila, lambertian lighting is done.

The constant in question represents how reflective the material is for a given part of the spectrum. A blue thing is something that reflects mostly blue wavelengths and so on. It is typically represented as a RGB colour value and in the fragment shader we have the variable '`materialDiffuse`' that represents this property (as we discussed in the first step):

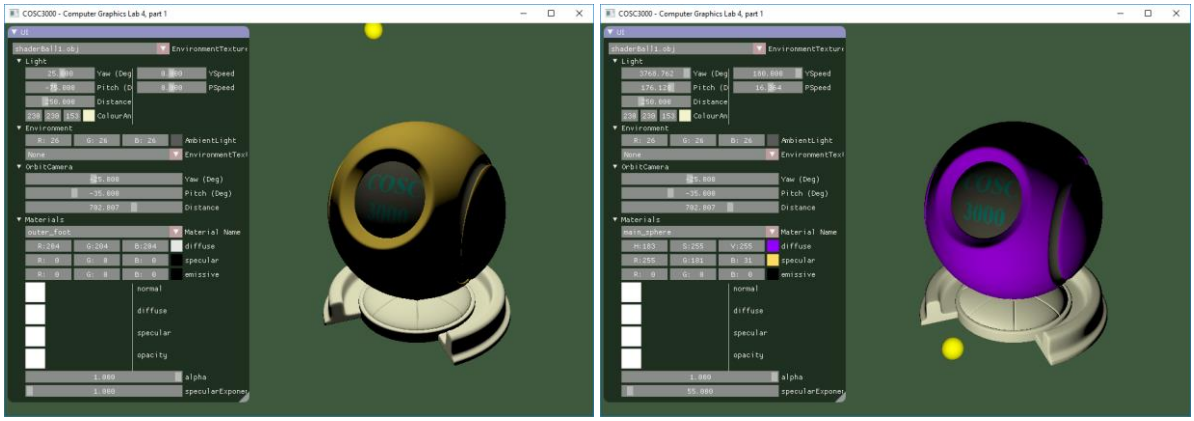
```
vec3 materialDiffuse = texture(diffuse_texture, v2f_texCoord).xyz * material_diffuse_color;
```

It is computed as the element-wise product of two factors, one of which is loaded from a texture, and the other is a parameter of the material. Both of these are configured through the .mtl file and can be tweaked in the UI.

To complete the lambertian shading model, all we need to do is multiply incoming light by the diffuse reflectance.

```
vec3 outgoingLight = incommingLight * materialDiffuse;
fragmentColor = vec4(outgoingLight, material_alpha);
```

The result should look like the below. We now see the influence of the material diffuse colours. In the UI, select the 'main_sphere' material and play with the diffuse reflectance. To see how the lighting changes turn on the light animation by setting the two light speeds to something nice.



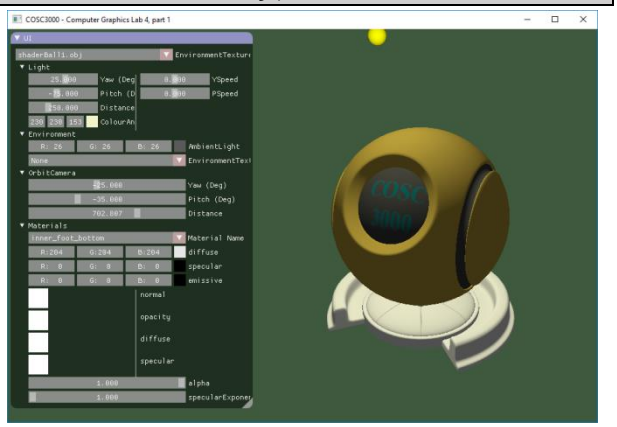
4. Ambient Light

Now, you might have noticed that it is very dark on the dark side of that moon [that's too small to be a moon! etc.[12]]. To fix that we need to take into account incoming light that did not arrive directly from the light source, i.e., *indirect light*. We will not try to solve this in anything remotely like an accurate way, but instead approximate all this with a single intensity/colour value.

This value is represented in the variable 'g_ambientLightColourAndIntensity' and is already set up as a uniform variable 'ambientLightColourAndIntensity' which is set by the python program. Since the ambient light has no directionality (it is from everywhere) it is simply added to the incoming light.

```
vec3 outgoingLight =
            (incommingLight + ambientLightColourAndIntensity) * materialDiffuse;
```

Note that we're adding it to the incoming light, and then applying the (constant) lambertian BRDF. This makes some sense, since this BRDF is also independent from the directions.

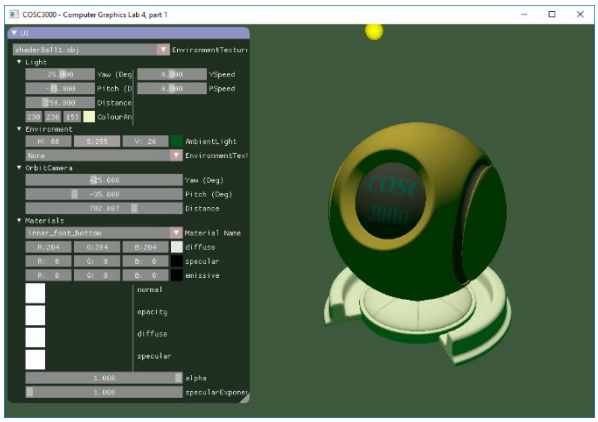


However, we will not do the same to the specular BRDF later since it makes little sense for a strongly directional effect. Keep in mind that this a very coarse approximation! The result should look like the image to the right:

You can tweak the level of ambient light in the UI

[12] https://xkcd.com/1458/

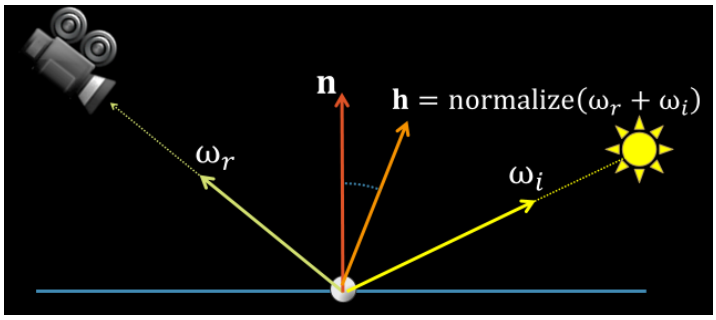
under ‘Environment/Ambient Light’. A strong tint can be useful if there is a strong colour dominating the environment. To the right I’ve set it to green (to sort of match the background).



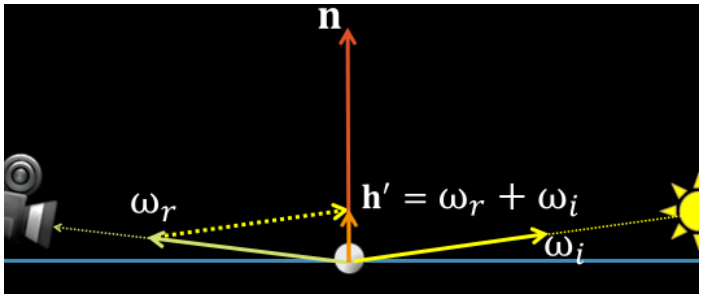
As an aside, a slightly more advanced ambient model, which is sometimes used for outdoor scenes is to have two colours and blend between them based on the orientation of the surface, such that things facing straight up get a blue ambient light (from the sky) and down a green tint (to represent grass reflecting light up). We will leave that for now though!

5. Specular lighting, part 1

Recall that specular lighting is the term used for light that is reflected (not absorbed an re-emitted like diffuse) off a *glossy* surface – i.e., something which is more rough than a mirror. This means that the effect is strongly directional. Exactly how to compute this directional influence is a matter under active research, in particular for real-time rendering. There are many models and we are using one which is reasonably simple and still based (or perhaps inspired by) physics.



As explained in the lecture the *Blinn-Phong* model uses what is known as the half-vector (or sometimes half-angle) to determine the strength of the reflection. This vector is simply the half-way direction between the incoming and outgoing directions. We can get it by simply adding these two vectors and normalizing the result. The second image to the right shows what happens if we don’t normalize. In general the resulting vector comes out shorter.

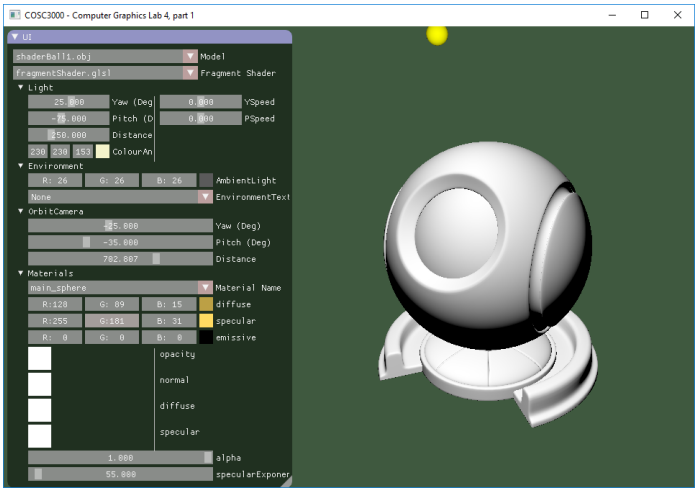


To calculate the half angle we need the outgoing direction, which is the direction towards the camera or eye position. In view space, this position is the origin, and thus the outgoing direction for a point P in view space as $\omega_r = \text{normalize}\big((0,0,0) - P\big)$. The below is the code to implement this:

```
vec3 viewSpaceDirToEye = normalize(-v2f_viewSpacePosition);
vec3 halfVector = normalize(viewSpaceDirToEye + viewSpaceDirToLight);
```

Using the half vector, add the following code to compute and output the base specular intensity, which is the cosine of the angle between the normal and the half vector. Note that we also need to clamp this to avoid any negative values.

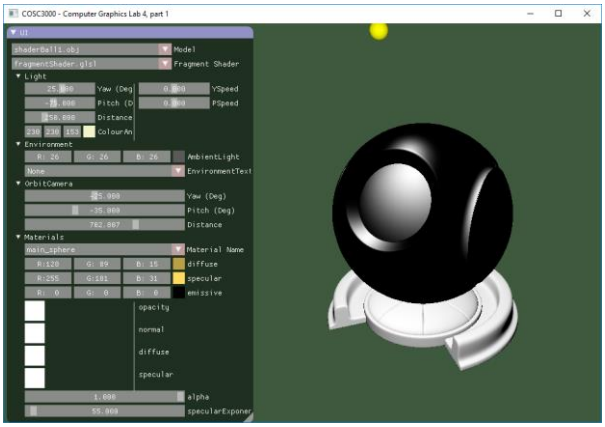
```
float specularIntensity = max(0.0, dot(halfVector, viewSpaceNormal));
fragmentColor = vec4(vec3(specularIntensity), material_alpha);
```



This gives a very broad highlight, and to control the size of the highlight to simulate more or less rough materials, we raise this quantity to some power, called the 'specular exponent'.

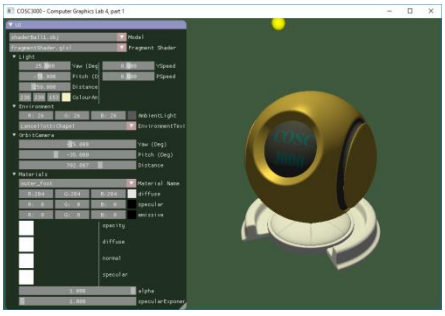
```
float specularIntensity =
      pow[13](max(0.0, dot(halfVector, viewSpaceNormal)),
material_specular_exponent);
fragmentColor = vec4(vec3(specularIntensity), material_alpha);
```

This changes the look on the main sphere drastically, with a much more concentrated highlight. As you can see in the UI the 'main_sphere' material has a specular exponent of 55 – corresponding to a much tighter highlight. Play around a bit with the exponents and angles to see the effect.



Now we can multiply this by the incoming light and the material specular colour to get a basic specular lighting model. The material specular reflectance is computed from the material parameters in the same way as the diffuse reflectance.

```
vec3 materialSpecular =
       texture(specular_texture, v2f_texCoord).xyz * material_specular_color;
vec3 outgoingLight =
        (incommingLight + ambientLightColourAndIntensity) * materialDiffuse +
         incommingLight * specularIntensity * materialSpecular;
fragmentColor = vec4(outgoingLight, material_alpha);
```

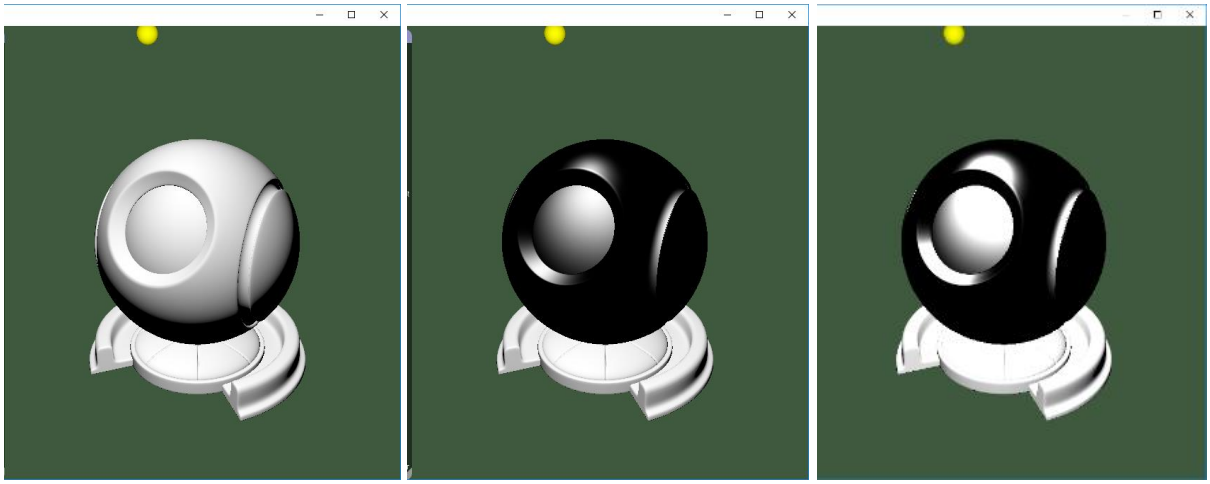


This involves all the basic components of a realistic lighting model, but we can with a few, relatively simple modifications make it more useful and easier to control. The first thing to notice is that the highlight is very weak looking. We'll look at this problem next.

[13] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/pow.xhtml

6. Normalizing the Blinn-Phong BRDF

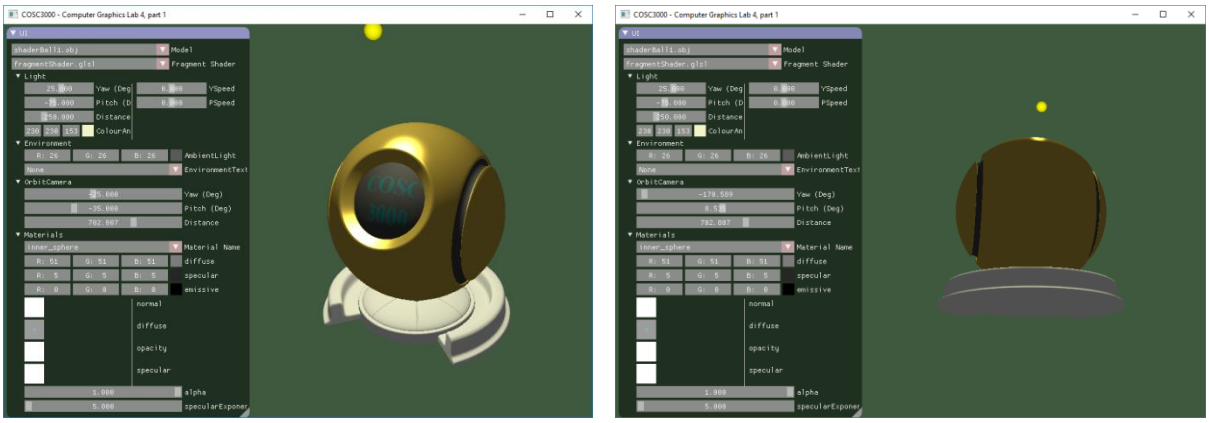
Compare the three images below:



The left image has a specular exponent of 1, and the middle has a specular exponent of 55. Intuitively, a higher exponent means that more light is reflected towards the eye, with less spread and thus should be stronger, more focussed, at the heart, this is an energy conservation issue. To compensate we introduce a normalization factor[14]:

```
float specularNormalizationFactor = ((material_specular_exponent + 2.0) / (2.0));
float specularIntensity = specularNormalizationFactor *
    pow(max(0.0, dot(halfVector, viewSpaceNormal)), material_specular_exponent);
fragmentColor = vec4(vec3(specularIntensity), material_alpha);
```

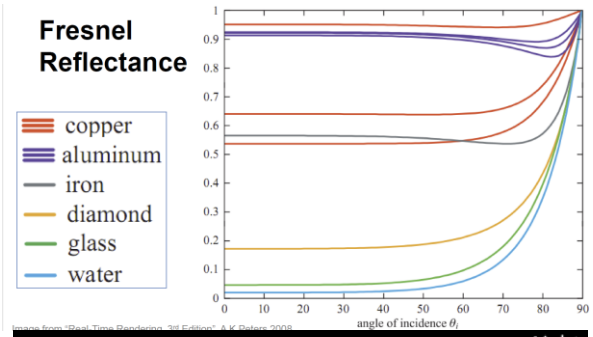
The result is shown in the rightmost image above. Plugging this into the full reflected light calculation we get:



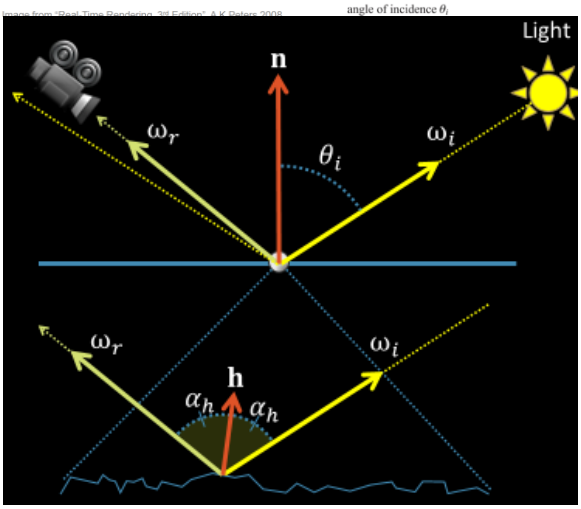
This is starting to look nice, but we will take one more step before we'll call the specular highlight good enough. The problem is shown in the right image.

[14] This can be done in a multitude of ways and deriving these factors is quite involved, You can read more in this blog post, if interested: http://www.thetenthplanet.de/archives/255

7. Fresnel reflection



Recall from the lecture that everything becomes a mirror at sufficiently glancing angles. Looking at the image to the right above, you can see that this is not happening at all! The highlight just fades away as the light goes behind. Play around with the light and camera position to see this for yourself.



We want to introduce this effect into the model. The parameter to the Fresnel effect is an angle. Which angle? For an optically flat surface it is the angle of incidence. For the incoming light this is the angle between the direction to the light and the half vector. Why? Recall that the half vector is the normal of the micro-facets that reflect light towards the camera. The others do not contribute at all, so we only care about the angle between incoming light and these.

The Schlick approximation to the Fresnel reflectance is very simple to compute. Note that the input we use is the cosine of the angle, since this is what we get from the dot product and is what we want anyway (rather than converting back and forth to angles, which involves trigonometry, and worse inverse trig).

```
vec3 fresnelSchick(vec3 r0, float cosAngle)
{
  return r0 + (vec3(1.0) - r0) * pow(1.0 - cosAngle, 5.0);
}
```

The input to this function is the reflectance when looking ‘straight on’ the surface, and we assume this is what the material specular colour represents. Thus we can calculate and visualize this.

```
vec3 fresnelSpecular = fresnelSchick(materialSpecular,
   max(0.0, dot(viewSpaceDirToLight, halfVector));
fragmentColor = vec4(fresnelSpecular, material_alpha);
```

The effect is fairly subtle for the material we have on the sphere, because it has a very strong ‘r0’, that is reflection when viewed straight on. Setting both ‘Material/diffuse’ and ‘Material/specular’ to (0,0,0) for the ‘main_sphere’ material in the UI makes it much more noticeable.

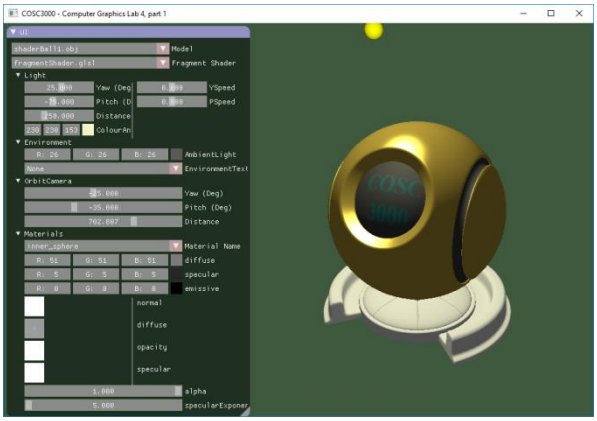
Without the fresnel reflectance the highlight would be gone.

Putting this all together concludes our lighting model, which I've called a normalized Blinn-Phong model, inspired by a true story (physics). The name is perhaps not a globally accepted one, but it contains the relevat keywords. The final calculations is as follows:

```
vec3 fresnelSpecular = fresnelSchick(materialSpecular,
                                max(0.0, dot(viewSpaceDirToLight, halfVector)));
vec3 outgoingLight =
          (incommingLight + ambientLightColourAndIntensity) * materialDiffuse +
           incommingLight * specularIntensity * fresnelSpecular;
fragmentColor = vec4(outgoingLight, material_alpha);
```

Experiment with the parameters of the materials to try to get a feeling for how to achieve different looks and materials.

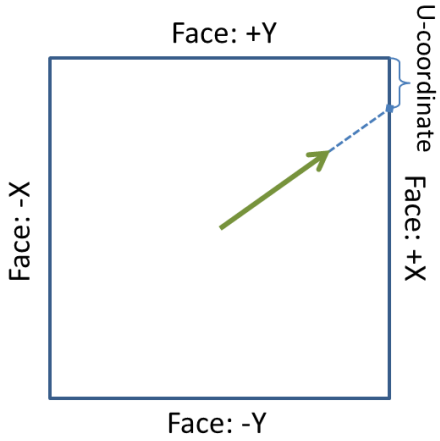
8. Mirror reflection



The material parameters on the main sphere thing on the shader ball are based somewhat on the measured parameters for 'gold'. With some good will, we might say that the result we have achieved so far looks goldy, a bit, very rough gold perhaps. The main problem is that most of the effect in a shiny metal comes from mirror-reflection at the surface, which our model does not capture at all.

Actually computing accurate reflections is not really possible in a real-time shader and generally speaking requires ray tracing. Thus we have to fake it. Happily, if all we are after is the sense that something is shiny, or mirror like, it doesn't matter that much what is shown in the reflection, just the way it moves when we change the view-point is a sufficient visual cue.

To get this change to be good enough, we need to calculate the reflection direction for the shading point, and then we perform a look-up in an *environment map* (or reflection map). This is a texture which represents the environment surrounding the object being rendered. There are many ways to do this, but the most straightforward way is to use a *cube-map texture.*



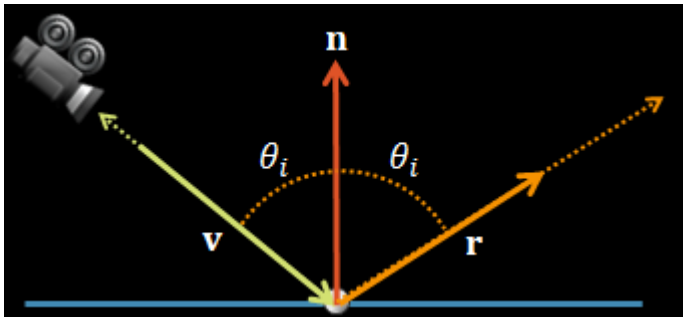
A cube map[15], like the name implies, is a collection of 6 2D-textures, that are considered sides of a cube. The cube map is sampled using a 3D direction vector, rather than a 2D coordinate, and the GPU maps this to a cube face, and then a 2D coordinate within that. A 2D-illustration of this is shown to the right, where the direction selects the '+X' face.

To calculate the reflection direction we just need the incoming direction, which is the direction *from* the camera, and the normal of the surface to reflect in. This is not exactly difficult to compute, but to make it even simpler GLSL provides a built in function 'reflect' to do this:

[15] https://www.khronos.org/opengl/wiki/Cubemap_Texture

```
vec3 viewSpaceReflectionDir = reflect(-viewSpaceToEyeDirection, viewSpaceNormal);
```

The calculation is illustrated to the right.



Using this we can perform a lookup in a cube map, like this, and output the colour to visualize the result.

```
vec3 envSample = texture(environmentCubeTexture, reflDirWorldSpace).xyz;
fragmentColor = vec4(envSample, material_alpha);
uniform samplerCube environmentCubeTexture;
```

For this to work we also need to declare the cube map sampler uniform. Add the following somewhere together with the other uniform declarations:

```
uniform samplerCube environmentCubeTexture;
```

And also, we must set the uniform to the correct texture stage and bind the texture. Add the following code in 'renderFrame' somewhere after 'glUseProgram(g_shader)' (but before the draw call!):

```
lu.setUniform(g_shader, "environmentCubeTexture", TU_EnvMap)
lu.bindTexture(TU_EnvMap, g_environmentCubeMap, GL_TEXTURE_CUBE_MAP)
```

The constant 'TU_EnvMap' is an integer constant defined in the Python program, and is chosen to avoid conflict with the texture units used by the ObjModel. 'g_environmentCubeMap' is the OpenGL id of the cube map currently selected in the UI. Running this should give the appearance shown to the right, selecting the 'NissiBeach2' environment texture in the UI. Rotate the camera around with the mouse, looks shiny!

But! (There is always a 'but'). This is actually not correct. It is not that easy to tell in this very artificial scene, but note how we only see one side of the reflection map. When rotating the camera it should change (now it looks like the camera is static and the object is rotated, which is not the intention). The reason is that we used the *view-space* reflection direction. If we want the environment to represent the environment in the world, then we must sample it in world space!

We'll fix this by adding another transformation matrix that takes a direction in view space to world space. Since we only plan on transforming direction vectors with this, a 3x3 matrix is sufficient. In 'renderFrame' add:

```
lu.setUniform(g_shader, "viewToWorldRotationTransform", \
                         lu.inverse(lu.Mat3(worldToViewTransform)))
```

And then in the fragment shader declare the uniform:

```
uniform mat3 viewToWorldRotationTransform;
```

Now we can perform the lookup in the correct space:

```
vec3 worldSpaceReflectionDir = viewToWorldRotationTransform *
                          reflect(-viewSpaceDirToEye, viewSpaceNormal);
vec3 envSample = texture(environmentCubeTexture, worldSpaceReflectionDir).xyz;
fragmentColor = vec4(envSample, material_alpha);
```

The visual difference is kind of hard to tell, since the environment that is being shown in the reflection is not present in the scene, but it should be clear that the change has achieved what we want. Below is a before and after comparison, with the camera pointing straight down.

View-space reflection direction	World-space reflection direction
	

Now. This is lovely if all we want to do is mirror balls, or other perfect mirrors. However, we want to integrate this effect together with the rest of the shading.

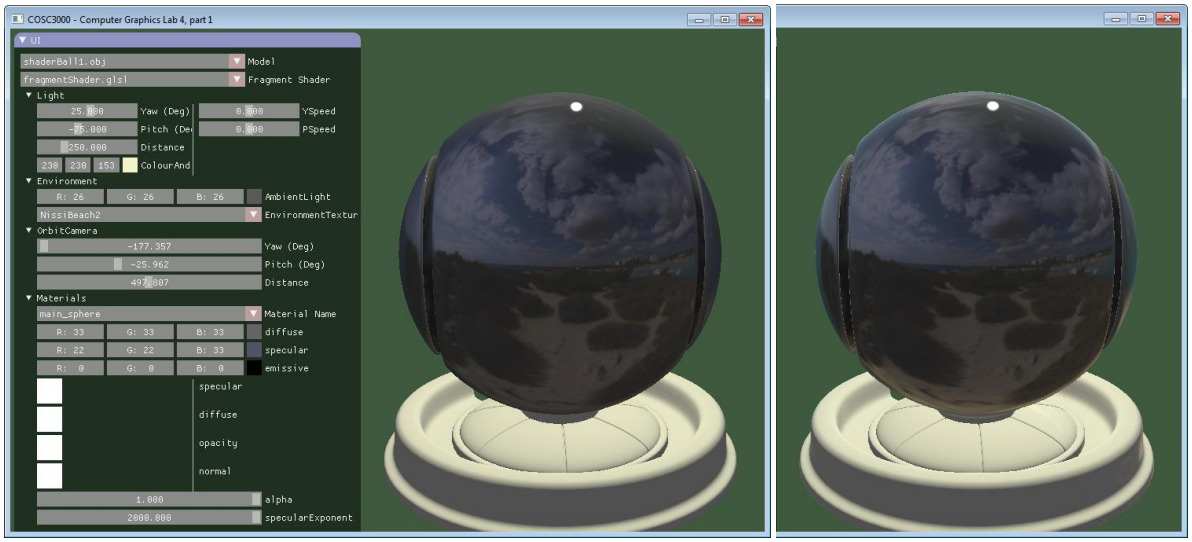
A first blush at this is to tint the reflection with the material specular colour and add it to the other outgoing light:

```
vec3 outgoingLight =
     (incommingLight + ambientLightColourAndIntensity) * materialDiffuse +
     incommingLight * specularIntensity * fresnelSpecular +
     envSample * materialSpecular;

fragmentColor = vec4(outgoingLight, material_alpha);
```

This is starting to look nice indeed, and rather more goldy than before!

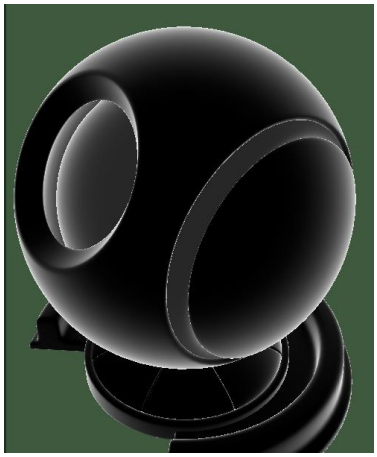
There is but one tiny detail left that would be nice to take a look at. Just using a constant reflection (the 'materialSpecular') irrespective of angle is not correct. For materials with near-mirror finish there is no difference. However, for materials that have a lower reflectivity we know from the Fresnel equations that it should be lower when looking straight on, as well as approaching 1 at glancing angles. Below I have changed both diffuse and specular reflection on the main sphere to a low gray value ('33' for all components in the UI).



Since we already have the tools handy (i.e., the 'fresnelSchick' function), this is a simple change:

```
vec3 fresnelSpecularEye =
  fresnelSchick(materialSpecular,
         max(0.0, dot(viewSpaceDirToEye, viewSpaceNormal)));

vec3 outgoingLight =
   (incommingLight + ambientLightColourAndIntensity) * materialDiffuse +
   incommingLight * specularIntensity * fresnelSpecular +
   envSample * fresnelSpecularEye;
```



The effect, shown above to the right, is fairly subtle, but you can see how the edges are much brighter as they reflect more light. Visualizing the '`fresnelSpecularEye`' with specular colour set to zero is shown to the right.

Basically we now have a different problem, which is that we can't get rid of reflections using the parameters we have. While this might make physically inspired *sense*, in our not-so physically based model we get into trouble. The problem really comes from the fact that we don't represent lighting with correct magnitudes. If we did, then direct light from the sun, for example, would easily drown out a subtle reflection from the environment. Since we don't have this we may be forced to introduce a non-physics based parameter to be able to tweak 'reflectiveness'. Another issue is that we can set both a high diffuse **and** a high specular with a low specular power. Essentially, we're then reflecting the same light twice(!), and a better model should not allow this. Intuitively, any given photon can be either absorbed and re-emitted (diffuse) **or** reflected at the surface (specular) but not both[16].

As stated before, these are all factors that a carefully crafted physically based model ought to take care of. Since this is a short course, we cannot go into these solutions here, and the industry is still in the process of working out how to best do this. A proper conversion to physically based rendering involves everything, from textures and lights and environment textures, to the shading calculations.

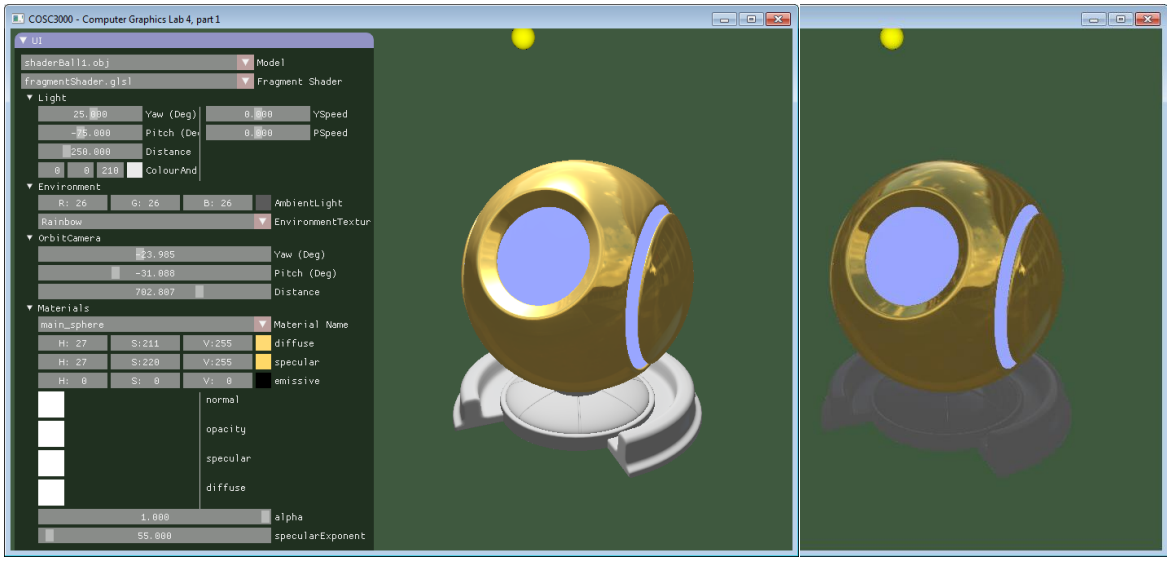
[16] Yeah, yeah, quantum whatever: not with 100% probability then, same difference.

9. Emissive materials

To round off the lab we'll add another common material parameter. This is known as the emissive colour and represents light that is emitted by the surface itself. It is typically assumed to emit equally in all directions, and since it is not reflected it is not affected by any BRDF. Thus, it can simply be added to the '`outgoingLight`' without much ado.

```
vec3 outgoingLight =
   (incommingLight + ambientLightColourAndIntensity) * materialDiffuse +
   incommingLight * specularIntensity * fresnelSpecular +
   envSample * fresnelSpecularEye +
   material_emissive_color;
```

In the material parameters provided by the ObjModel we have '`material_emissive_color`' representing this quantity, and there is no associated texture.



In the above the 'inner_sphere' has been set to have an emissive colour, and nothing else. The result is that it has the same colour and intensity no matter the incoming light. This is useful to create things such as lamps, which should always appear bright – for example lights on cars etc. Note that unless somehow added to the rendering (and this is a pretty big somehow!) this 'light' does not affect anything else in the scene unless we place a light source to achieve that.

Appendix

Colour spaces, sRGB versus Linear RGB

The keen observer might have noticed that the GUI looks a bit differently coloured from previous labs. This is because we have changed the render target to be interpreted as being in the sRGB colour space[17]. We do this (in the 'magic') by asking glfw to create a 'sRGB capable' default framebuffer for the window:

```
glfw.window_hint(glfw.SRGB_CAPABLE, 1)
```

And then in the program we tell OpenGL that we want the transformation to sRGB to be applied to all fragment colours as they are merged into the framebuffer (in '`initResources`'):

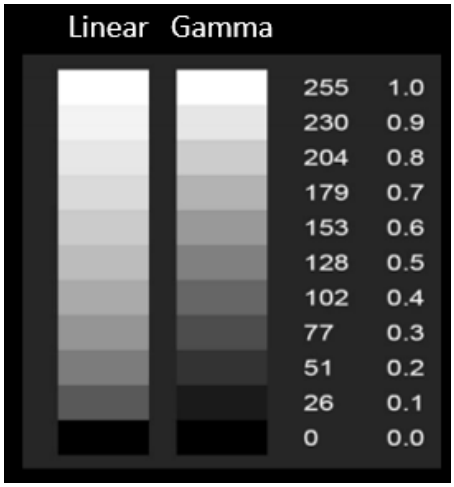
```
glEnable(GL_FRAMEBUFFER_SRGB)
```

Thus our shaders (and any other colour, such as the glClearColour) is interpreted as being in linear colour space, and since the frame buffer is interpreted as being in sRGB, a conversion occurs, which is roughly equivalent to:

```
vec3 toSrgb(vec3 color)
{
  return pow(color, vec3(1.0 / 2.2));
}
```

Which is in fact what the code used to do before I went and turned on the OpenGL built-in features to do the conversion automatically.

The reason this is needed is that most digital images such as photos, and (importantly) ***monitors***, use the *sRGB* by default as it matches the human perceptive system better – a linear colour space wastes too much precision for almost-white colours. Computing lighting in sRGB space makes no mathematical sense whatsoever, so we have taken care (in the ObjModel.loadTexture function) to flag the relevant textures as being in sRGB space, such that they are automatically converted to linear when we sample them. Textures that store ***data*** other than colours are not usually represented in sRGB and should not be converted. For example, the opacity texture which represents how opaque the surface is. The ObjModel takes care of this too.



Also note that the skeleton code for the mega racer uses the explicit conversion in the shader.

[17] https://en.wikipedia.org/wiki/SRGB