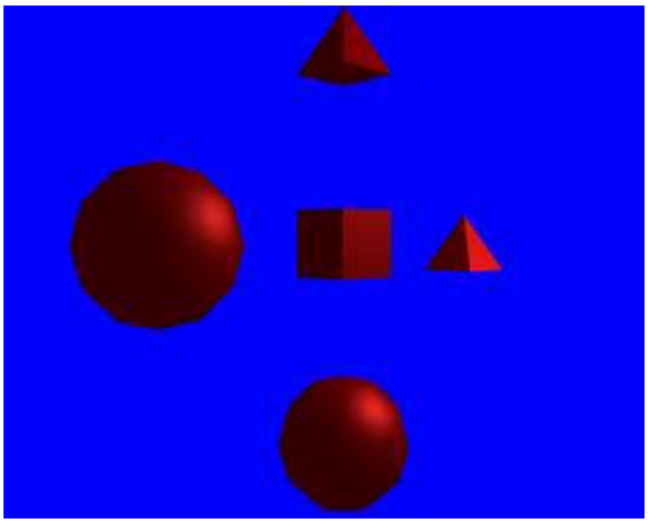
Tutorial 3: Lighting and Materials

This tutorial covers how to add lighting and high quality materials with OpenGL.



This tutorial will continue on from the last by extending the previously created code. It is assumed that you have created a copy of last tutorial's code that can be used for the remainder of this one.

Part 1: Object Normals

Lighting and material equations all require the surface normal to be known during calculation. Previous tutorials have only worked on vertex positions so in order to use lighting equations additional normal information needs to be added and passed to the vertex shader. Additional data can be passed to the vertex stage in one of a number of ways. One way is to create an additional buffer that just contains vertex normal information and then pass that to the shader inputs. Another way is to combine the normal information with the vertex position in the same buffer. The best performing way to do this is to group each vertices position and normal together (i.e. position1, normal1, position2, normal2, etc.). This improves performance by maintaining data locality which improves memory performance.

1. To add normal information, we need to modify the functions we previously made to generate geometry. The first thing we must do is add vertex normal data to out custom vertex type.

```
struct CustomVertex
{
    vec3 v3Position;
    vec3 v3Normal;
};
```

An unfortunate side effect of using per-vertex normal is that index buffers loose some of their efficiency as vertices shared between disjoint faces must now be separated. This is because although several geometry faces may share the same vertices each of those faces may now have a different normal requiring them to be separated. This mostly effects any object with hard edges. However modern GPUs are optimised for index buffer usage so despite the loss in efficiency they are still the recommended approach.

2. The "GL_GenerateCube" function needs to be updated so that it creates vertex normal information. Since a cube has entirely hard edges it means that no vertex can be shared anymore as each of them needs a new normal corresponding to the direction vector perpendicular to each face. Therefore, there now needs to be 26 vertices made in order to describe a cube.

```
CustomVertex VertexData[] = {
    // Create back face
    { vec3( 0.5f,  0.5f, -0.5f), vec3( 0.0f,  0.0f, -1.0f) },
    { vec3( 0.5f, -0.5f, -0.5f), vec3( 0.0f,  0.0f, -1.0f) },
    { vec3(-0.5f, -0.5f, -0.5f), vec3( 0.0f,  0.0f, -1.0f) },
    { vec3(-0.5f,  0.5f, -0.5f), vec3( 0.0f,  0.0f, -1.0f) },
    // Create left face
    { vec3(-0.5f,  0.5f, -0.5f), vec3(-1.0f,  0.0f,  0.0f) },
    { vec3(-0.5f, -0.5f, -0.5f), vec3(-1.0f,  0.0f,  0.0f) },
    { vec3(-0.5f, -0.5f,  0.5f), vec3(-1.0f,  0.0f,  0.0f) },
    { vec3(-0.5f,  0.5f,  0.5f), vec3(-1.0f,  0.0f,  0.0f) },
    // Create bottom face
    { vec3( 0.5f, -0.5f, -0.5f), vec3( 0.0f, -1.0f,  0.0f) },
    { vec3( 0.5f, -0.5f,  0.5f), vec3( 0.0f, -1.0f,  0.0f) },
    { vec3(-0.5f, -0.5f,  0.5f), vec3( 0.0f, -1.0f,  0.0f) },
    { vec3(-0.5f, -0.5f, -0.5f), vec3( 0.0f, -1.0f,  0.0f) },
    // Create front face
    { vec3(-0.5f,  0.5f,  0.5f), vec3( 0.0f,  0.0f,  1.0f) },
    { vec3(-0.5f, -0.5f,  0.5f), vec3( 0.0f,  0.0f,  1.0f) },
    { vec3( 0.5f, -0.5f,  0.5f), vec3( 0.0f,  0.0f,  1.0f) },
    { vec3( 0.5f,  0.5f,  0.5f), vec3( 0.0f,  0.0f,  1.0f) },
    // Create right face
    { vec3( 0.5f,  0.5f,  0.5f), vec3( 1.0f,  0.0f,  0.0f) },
    { vec3( 0.5f, -0.5f,  0.5f), vec3( 1.0f,  0.0f,  0.0f) },
    { vec3( 0.5f, -0.5f, -0.5f), vec3( 1.0f,  0.0f,  0.0f) },
    { vec3( 0.5f,  0.5f, -0.5f), vec3( 1.0f,  0.0f,  0.0f) },
    // Create top face
    { vec3( 0.5f,  0.5f,  0.5f), vec3( 0.0f,  1.0f,  0.0f) },
    { vec3( 0.5f,  0.5f, -0.5f), vec3( 0.0f,  1.0f,  0.0f) },
    { vec3(-0.5f,  0.5f, -0.5f), vec3( 0.0f,  1.0f,  0.0f) },
    { vec3(-0.5f,  0.5f,  0.5f), vec3( 0.0f,  1.0f,  0.0f) },
};
```

3. Next the index buffer needs updating so that it corresponds to the changed vertex list data. Since there are now no longer any shared vertices between each face the index buffer becomes mostly a sequential index into the vertex buffer.

```
GLuint uiIndexData[] = {
     0,  1,  3,  3,  1,  2,  // Create back face
     4,  5,  7,  7,  5,  6,  // Create left face
     8,  9, 11, 11,  9, 10,  // Create bottom face
    12, 13, 15, 15, 13, 14,  // Create front face
    16, 17, 19, 19, 17, 18,  // Create right face
    20, 21, 23, 23, 21, 22   // Create top face
};
```

Generating vertex normals for a sphere is rather simple. The vertex normal is simply the direction from the current vertex from the spheres centre (normal = position – centre). This is even simpler assuming the sphere is centred at the origin (0,0,0) as then the vertex normal is just the position.

4. The code in "GL_GenerateSphere" now needs to be updated so that it generates vertex normals for the sphere. Since the vertex normal is simply just the position this just requires an extra step in order to set the normal to the same values as the vertex. In the function "GL_GenerateSphere" add an extra line under each location that a vertex position is calculated. This line should add the vertex normal. As a sphere shares all its vertices between each face then the index buffer does not to be updated.

```
// Create vertex
p_vBuffer->v3Position = vec3(fX, fY, fZ);
p_vBuffer->v3Normal = vec3(fX, fY, fZ);
```

5. In the last tutorial it was also required that you create your own function to generate geometry for a square pyramid. Update the "GL_GeneratePyramid" function so that it also creates appropriate geometry normals.

6. Now that the data in the vertex buffer has been modified the next step is to notify OpenGL of the changed vertex buffer layout by using "glVertexAttribPointer". As there is now normal information for each vertex then the stride for each position is now altered as the size between the start of each vertex position is now twice as big (due to the inclusion of the normal). This conveniently is already handled by using the "sizeof" function to determine the stride. We then need to add a second vertex attribute for use with the new vertex normal data which we will specify at location '1'. Replace each of the previous vertex attribute calls with the required updates (remember to replace all of them).

```
// Specify location of data within buffer
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(CustomVertex),
(const GLvoid *)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(CustomVertex),
(const GLvoid *)offsetof(CustomVertex, v3Normal));
glEnableVertexAttribArray(1);
```

7. The first step in using the new vertex normal data is to add the corresponding input to the vertex shader code. Since we specified the vertex normal data is attribute '1' it is important to make sure the corresponding layout "layout(location)" qualifier matches.

```
layout(location = 1) in vec3 v3VertexNormal;
```

8. It should be possible to now run the program. Of course so far we haven't done anything with the vertex data but at the very least it is now possible to make sure that the position information hasn't been harmed during the modifications. Running the program should result in the same output as the last tutorial.

Part 2: Per-Vertex Lighting

Adding lighting and shading calculation to a render system requires adding the appropriate code into the shaders. Traditional OpenGL render systems have Vertex as well as Fragment shaders, the choice of which shader the code is added to has notable effects on the final quality and performance. Adding the material code in the Vertex shader therefore means that the lighting equation is performed on each vertex. This is called per-vertex lighting or Gouraud shading. In per-vertex lighting the shaded colour due to the material and lighting properties is calculated for each vertex and then output. The output is then interpolated by the renderer and passed into the Fragment shader. This interpolation stage is responsible for blending each output colour from each vertex in a triangle to determine the colour for each pixel that the triangle covers.

In order to perform lighting operations, it is required to determine how much light is available in the scene. This requires modelling light sources and adding them accordingly. Traditional lighting systems have used 3 different abstract types used to represent different lighting situations; point lights, spot lights and directional lights. The most obvious and commonly used is the point light. A point light is simply a point in space that emits light uniformly in all directions around it. A point light can be described by its position, the amount of light that it emits (i.e. colour and intensity) and a value used to describe the falloff of the light. The falloff is used to determine how the lights brightness diminishes the further away from the light that you are.

9. To add lighting to our test scene we must first add some lights. We will do this by adding a single point light. A point light is defined by its position, intensity and falloff coefficient so these need to be added to the vertex shader code. OpenGL allows data to be grouped together using a '`struct`'. This functions identically to a '`struct`' in the C programming language. We will add a structure called "`PointLight`" to describe each of the values of a point light. Next we will then create a UBO input that contains a single instance of "`PointLight`" which we will use "`layout(binding)`" to bind it to the next available UBO location (in this case '2').

```
struct PointLight {
    vec3 v3LightPosition;
    vec3 v3LightIntensity;
    float fFalloff;
};
layout(binding = 2)
uniform PointLightData {
    PointLight PointLights;
};
```

Traditionally most renderers have used the Phong material shader to represent material properties. This shader is responsible for determining the amount of visible light (i.e. colour and intensity) visible from a point on a surface due to any incoming light that falls on that point. An improved version of this shader was proposed by Blinn and is often referred to as the Blinn-Phong shader. The visible light $L_V(\vec{\boldsymbol{v}})$ along view direction $\vec{\boldsymbol{v}}$ from a point based on the incoming light $E_L(\vec{\boldsymbol{l}})$ from a single light with incoming direction $\vec{\boldsymbol{l}}$ can be approximated by the modified Blinn-Phong equation such that:

$$L_V(\vec{\boldsymbol{v}}) = [\rho_d\overline{(\hat{\vec{\boldsymbol{n}}}\cdot\hat{\vec{\boldsymbol{l}}})} + \rho_s\overline{(\hat{\vec{\boldsymbol{n}}}\cdot\vec{\boldsymbol{h}})}^{n}]\overline{(\hat{\vec{\boldsymbol{n}}}\cdot\hat{\vec{\boldsymbol{l}}})}E_L(\vec{\boldsymbol{l}})$$

Where $\vec{\boldsymbol{n}}$ is the normal to the surface at the visible point and ρ_d is the albedo (colour) of the diffuse component of light, ρ_s is the equivalent for specular light and n is used to describe the shininess of the surface. The half-vector $\vec{\boldsymbol{h}}$ is defined as the direction vector in between the light and view direction vectors such that:

$$\vec{\boldsymbol{h}} = \frac{\vec{\boldsymbol{v}} + \vec{\boldsymbol{l}}}{\|\vec{\boldsymbol{v}} + \vec{\boldsymbol{l}}\|}$$

10. To use the Blinn-Phong material shader we need to add the appropriate data inputs to our vertex shader. To do this we need to add another UBO that contains the required data for a Blinn-Phong material. A Blinn-Phong material can be described by its diffuse albedo and its specular as well as a value for its shininess. For the moment we will use "`fRoughness`" to describe the shininess value (this will make more sense later on). We again make sure to explicitly bind it to the next available UBO location (in this case '3').

```
layout(binding = 3)
uniform MaterialData {
    vec3  v3DiffuseColour;
    vec3  v3SpecularColour;
    float fRoughness;
};
```

11. Since we need to know the view direction we therefore need to know where the camera actually is. This requires updating the input camera data to also include the cameras position. We can then use this position to calculate the view direction from the camera to the current point.

```
uniform CameraData {
    mat4 m4ViewProjection;
    vec3 v3CameraPosition;
};
```

Outputs from the vertex shader stage are interpolated by the OpenGL render system. The type of interpolation is determined by the qualifiers applied to the specific output variable. Output variables with the "`flat`" qualifier are passed through to the Fragment shader stage exactly as they are without any modification. Variables marked with "`smooth`" are set to be interpolated between each vertex based on the position being covered by the corresponding fragment in the fragment Shader. This interpolation is performed in the final ViewProjection space so that it is perspectively smooth with respect to the output image.

12. To pass the calculated shaded value for the vertex to the fragment shader we need to add an output to the vertex shader. This output should be marked with "`smooth`" to specify it should be interpolated across the visible triangle face. We will also add a "`layout(location)`" qualifier that we can use to link the output of the Vertex shader with the input of the Fragment shader (note input and output locations are separate so the same location ID can be used in either). OpenGL is also capable of automatically linking input and output variables that have an identical name. However, in this case we want to explicitly handle the linking so we can control how the inputs/outputs are passed. So as we have a single colour output we will link it to the '0' location.

```
layout(location = 0)
smooth out vec3 v3ColourOut;
```

13. Now that we have all the inputs and outputs setup in our vertex shader the next piece is to add the implementation of the modified Blinn-Phong shading model. To do this we will create a new GLSL shader function that we will call "`blinnPhong`". GLSL shader functions are much like standard C functions. The main difference is that parameters passed into the shader should be specified by their direction where "`in`" specifies data coming into the shader, "`out`" can be used to pass data out and "`inout`" to pass data both ways. The only output for our shader is the final calculated colour which can be returned directly from the shader as a "`vec3`". As GLSL provides many inbuilt functions to perform certain operations we will use them here as they often correspond to optimised instructions used on GPU hardware. Since the half-vector must be normalized before use we can use the "**`normalize`**" function to handle that for us. GLSL also has an inbuilt function to handle dot product operations that is called "**`dot`**". Since it's possible for the normal and half vector to be pointing in opposite directions then it is possible that the result will be a negative value.

Negative values are incorrect so we handle these by setting them to zero. This can be done by the GLSL inbuilt function “**max**” which we use to return ‘0’ if the result of the dot product is less than zero. Finally, to calculate the power term we can use the GLSL inbuilt “**pow**” which can be used to determine the first input to the power of the second.

```
vec3 blinnPhong(in vec3 v3Normal, in vec3 v3LightDirection,
in vec3 v3ViewDirection, in vec3 v3LightIrradiance, in vec3 v3DiffuseColour,
in vec3 v3SpecularColour, in float fRoughness)
{
    // Get diffuse component
    vec3 v3Diffuse = v3DiffuseColour;

    // Calculate half vector
    vec3 v3HalfVector = normalize(v3ViewDirection + v3LightDirection);

    // Calculate specular component
    vec3 v3Specular = pow(max(dot(v3Normal, v3HalfVector), 0.0f), fRoughness) *
v3SpecularColour;

    // Combine diffuse and specular
    vec3 v3RetColour = v3Diffuse + v3Specular;

    // Multiply by view angle
    v3RetColour *= max(dot(v3Normal, v3LightDirection), 0.0f);

    // Combine with incoming light value
    v3RetColour *= v3LightIrradiance;
    return v3RetColour;
}
```

Point lights are described by their position, light intensity and a coefficient for the light falloff. To correctly model the light then an equation that models this light falloff must be used. Physically correct falloff from a point light radiating in all directions is modelled by a quadratic inverse function with respect to the distance r from the light source. This falloff is often scaled by a value k_r that is used to scale the falloff to achieve a desired effect. The falloff $f_{dist}(r)$ for a light can then be modelled by the function:

$$f_{dist}(r) = \frac{1}{k_r r^2}$$

14. In order to correctly model light falloff, we need to add an additional function to our Vertex shader. We will call this function “`lightFalloff`” and it will be responsible for determining the light falloff for a specified input distance and light parameters. GLSL has a convenient inbuilt function “**`distance`**” that we can use to calculate the distance between 2 points. The final light value is then returned from the function in a “`vec3`”.

```
vec3 lightFalloff(in vec3 v3LightIntensity, in float fFalloff,
in vec3 v3LightPosition, in vec3 v3Position)
{
    // Calculate distance from light
    float fDist = distance(v3LightPosition, v3Position);

    // Return falloff
    return v3LightIntensity / (fFalloff * fDist * fDist);
}
```

The final step is to now use the light falloff function and the Blinn-Phong function to determine the final colour contribution for the current vertex. This requires adding additional code to our vertex shader. As each input direction should be normalized we will do that here using the GLSL inbuilt function "`normalize`" before passing them to the appropriate functions.

```
void main()
{
    // Transform vertex
    vec4 v4Position = m4Transform * vec4(v3VertexPos, 1.0f);
    gl_Position = m4ViewProjection * v4Position;

    // Transform normal
    vec4 v4Normal = m4Transform * vec4(v3VertexNormal, 0.0f);

    // Normalize the inputs
    vec3 v3Position = v4Position.xyz / v4Position.w;
    vec3 v3Normal = normalize(v4Normal.xyz);
    vec3 v3ViewDirection = normalize(v3CameraPosition - v3Position);
    vec3 v3LightDirection = normalize(PointLights.v3LightPosition -
v3Position);

    // Calculate light falloff
    vec3 v3LightIrradiance = lightFalloff(PointLights.v3LightIntensity,
PointLights.fFalloff, PointLights.v3LightPosition, v3Position);

    // Perform shading
    vec3 v3RetColour = blinnPhong(v3Normal, v3LightDirection, v3ViewDirection,
v3LightIrradiance, v3DiffuseColour, v3SpecularColour, fRoughness);
    v3ColourOut = v3RetColour;
}
```

15. Now that the Vertex shader is done the next step is to update the Fragment shader. This is rather simple as all the Fragment shader does is output the passed in colour value from the Vertex shader. To use this, we need to add the corresponding input to the Fragment shader. Since in the Vertex shader the corresponding output was marked at location '0' then we must mark the input to '0' as well using "`layout(location)`". The Fragment shader then has nothing else to do but pass the input colour directly to the output.

```
layout(location = 0) in vec3 v3ColourIn;
out vec3 v3ColourOut;

void main()
{
    v3ColourOut = v3ColourIn;
}
```

16. The shader code is now complete so the next step is to add the corresponding host code to pass the required data into the shader program. The first requirement is to add the light data. As there is only one point light being used then we simply need to declare the point light data and then create a UBO to store it. Just like the shader code a point light is described by its position, intensity and falloff coefficient. As standard the OpenGL shader code expects "`vec3`"'s to be passed in with an alignment of 16B. Since a normal "`vec3`" is only 12B then any trailing

```
struct PointLightData
{
    aligned_vec3 m_v3Position;
    aligned_vec3 m_v3Colour;
    float        m_fFalloff;
};
GLuint g_uiPointLightUBO;
```

"vec3"'s will have the wrong start alignment. To fix this we can use a special GLM type "aligned_vec3" that ensures that the vector has the correct alignment.

17. Now we must create the corresponding material declaration as well. Just like in the shader code a material is defined by its diffuse and specular component as well as a roughness coefficient. We then need to create the declaration for a UBO used to pass this material data to the shader.

```
struct MaterialData
{
    aligned_vec3 v3DiffuseColour;
    aligned_vec3 v3SpecularColour;
    float        fRoughness;
};
GLuint g_uiMaterialUBO
```

18. Now we must pass the actual light data to the shader by adding code to the "GL_Init" function to fill the corresponding UBO. We will create a light at position (6, 6, 6) and set its intensity to a uniform white colour with a falloff of 0.01. Copying data into the UBO and then binding that UBO is performed identically to the way we have handled it previously. The only difference is that we will bind the Light data to location '2'. As the light data will not change then we can link the UBO to binding location '2' immediately.

```
// Create light UBO
glGenBuffers(1, &g_uiPointLightUBO);
PointLightData Light = {
    vec3( 6.0f, 6.0f, 6.0f ),      // Light position
    vec3( 1.0f, 1.0f, 1.0f ),      // Light colour
    0.01f };
glBindBuffer(GL_UNIFORM_BUFFER, g_uiPointLightUBO);
glBufferData( GL_UNIFORM_BUFFER, sizeof( PointLightData ), &Light,
GL_STATIC_DRAW );

// Bind Light UBO
glBindBufferBase(GL_UNIFORM_BUFFER, 2, g_uiPointLightUBO);
```

19. Next we need to do the same thing to load the material data. Here we will create a red material by setting the diffuse component to red (1, 0, 0). We will then set the specular value to a redy-white colour. We will then set the specular roughness to 15. Just like with the light data we will link it with the next available binding location (here it is 3) and then as we won't be changing the data during runtime then the material UBO can be immediately bound.

```
// Create material UBO
glGenBuffers(1, &g_uiMaterialUBO);
MaterialData Material = {
    vec3(1.0f, 0.0f, 0.0f),    // Diffuse colour
    vec3(1.0f, 0.3f, 0.3f),    // Specular colour
    15.0f};                    // Roughness
glBindBuffer(GL_UNIFORM_BUFFER, g_uiMaterialUBO);
glBufferData(GL_UNIFORM_BUFFER, sizeof(MaterialData), &Material,
GL_STATIC_DRAW);

// Bind material UBO
glBindBufferBase(GL_UNIFORM_BUFFER, 3, g_uiMaterialUBO);
```

20. Since we have created and initialised additional UBOs for the light and material then you should also add the appropriate functions to "`GL_Quit`" to handle cleaning up these buffers.

21. The last step is to update the camera data being passed to the renderer by adding in the additional camera position. Currently the camera data has just been a single 4x4 matrix which was loaded directly. Now we want to combine multiple data elements so we need to create a new camera structure.

```
struct CameraData
{
    aligned_mat4 m_m4ViewProjection;
    aligned_vec3 m_v3Position;
};
```

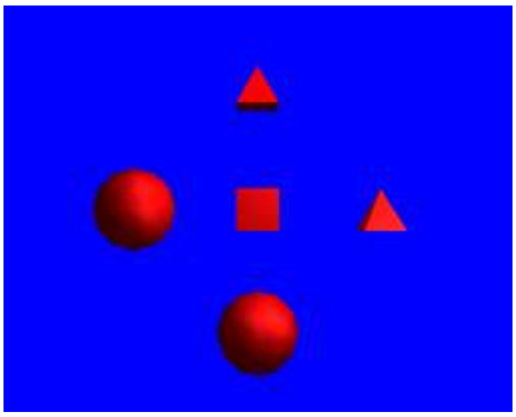
22. We then modify the update function so that it passes the new combined camera data instead of just the view projection matrix.

```
// Create updated camera data
CameraData Camera = {
    m4ViewProjection,
    g_CameraData.m_v3Position
};

// Update the camera buffer
glBindBuffer(GL_UNIFORM_BUFFER, g_uiCameraUBO);
glBufferData(GL_UNIFORM_BUFFER, sizeof(CameraData),
&Camera, GL_DYNAMIC_DRAW);
```

23. Assuming everything went correctly you should now be able to run the program and see the effects of the new lighting and shading code. At the moment though we can only see the object on the side with the light. One way to approximate surrounding lighting effect is to add an ambient light value. This value is simply multiplied by the surfaces diffuse value to provide a fixed amount of lighting irrespective of the current lighting conditions. Compare the difference with adding in a fixed ambient value at the very end of the shader code before the return statement.

```
// Add in ambient contribution
v3RetColour += v3DiffuseColour * vec3(0.3f);
```



The traditional Blinn-Phong lighting model is not energy conserving and does not satisfy Helmholtz reciprocity. This makes it potentially broken when using in more complicated lighting models. A fix for some of these issues is to use the Normalized version of the model. Normalizing however changes the way the model behaves and will result in materials looking differently than intended if they were made for the un-normalized model. Normalizing the Blinn-Phong model also changes the behaviour of the specular coefficients as now the value of n can be used to directly model the surface roughness and therefore becomes a more important value than ρ_s. However, it does have the advantage that it can model different types of surfaces easier as the ρ_s and n coefficients behave much more like real world values. This gives content creators much more flexibility and control when creating material data while also resulting in higher quality visual output. The normalized modified Blinn-Phong equation is:

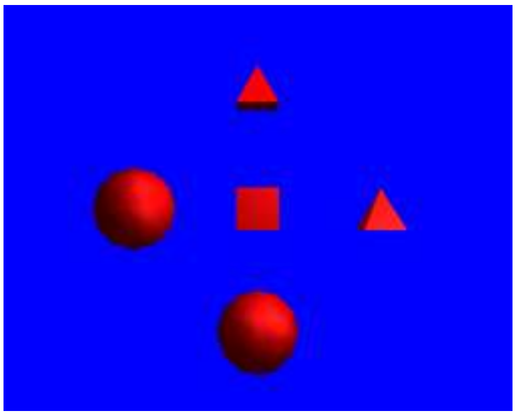
$$L_V(\vec{\boldsymbol{v}}) = [\frac{1}{\pi}\rho_d(\overline{\hat{\vec{\boldsymbol{n}}} \cdot \hat{\vec{\boldsymbol{l}}}}) + \frac{n+8}{8\pi}\rho_s\overline{(\hat{\vec{\boldsymbol{n}}} \cdot \vec{\boldsymbol{h}})}^{n}\,](\overline{\hat{\vec{\boldsymbol{n}}} \cdot \hat{\vec{\boldsymbol{l}}}})E_L(\vec{\boldsymbol{l}})$$

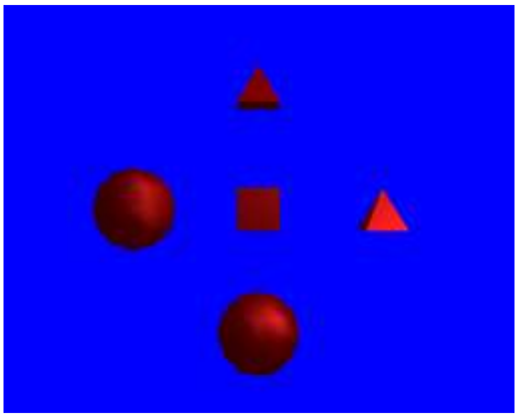
24. Add normalization code to your existing Blinn-Phong shader. This simply requires modifying the existing calculated diffuse and specular values by the appropriate normalization values added into the normalized Blinn-Phong equation. To do this the diffuse component needs to be multiplied by $\frac{1}{\pi}$ and the specular component needs to be multiplied by $\frac{n+8}{8\pi}$. Also the choice of using the name "`fRoughness`" should now make more sense. In order to make the added code work you will need to add defines for `M_RCPPI` and `M_PI` yourself. This is done in an identical manor to C code by adding the defines to the top of the shader code (but after `#version`).

```
// Normalize diffuse and specular component
v3Diffuse *= M_RCPPI;
v3Specular *= (fRoughness + 8.0f) / (8.0f * M_PI);
```

```
#define M_RCPPI 0.31830988618379067153776752674503f
#define M_PI 3.14159265358979323846264338327950f
```

25. You should now be able to run the program and observe the difference made by adding the normalization values.





Part 3: Per-Fragment Lighting

Per-vertex lighting has the downside that lighting calculations are only performed on each vertex of a surface. When a single triangle occupies many pixels in the final image using per-vertex lighting has a notable performance advantage as lighting calculations from each vertex can be used to shade multiple fragments. However, the disadvantage is that any lighting effects occurring on pixels that cover areas in between each vertex is completely skipped. This can cause disjoint lighting especially when either the light, camera or object are moving. Per-fragment lighting fixes this issue by performing lighting calculations in the Fragment shader instead of the Vertex shader. This allows for lighting smaller than a single triangle to be detected. This type of lighting is also called Phong lighting (not to be confused with the Phong shader).

26. To use per-fragment lighting you now need to make a new vertex and fragment shader. As a base you can copy the existing shaders and then just move and replace code as needed. Since the shading is now to be performed in the Fragment shader then the buffer blocks for the light and material data can now be moved to the Fragment shader. The "`blinnPhong`" and "`lightFalloff`" functions should also be moved into the Fragment shader.

27. Since we need the camera position in order to work out the view direction then we also need a copy of the camera data in the Fragment shader. Since uniform buffer blocks can be passed to multiple shaders at once all we have to do is copy the "`CameraData`" uniform to the Fragment shader.

28. Next we need to move the code to perform the actual shading operations from the Vertex to Fragment shaders. The only code remaining in the Vertex shader should be the code to calculate the position and the normal. As these values are needed in the Fragment shader for shading calculations then instead of outputting a colour value from the Vertex shader we instead need to output the transformed position and normal. The final Vertex shader code should only consist of the transformations and corresponding output of position and normal.

```
layout(location = 0) smooth out vec3 v3PositionOut;
layout(location = 1) smooth out vec3 v3NormalOut;

void main()
{
    // Transform vertex
    vec4 v4Position = m4Transform * vec4(v3VertexPos, 1.0f);
    gl_Position = m4ViewProjection * v4Position;
    v3PositionOut = v4Position.xyz;

    // Transform normal
    vec4 v4Normal = m4Transform * vec4(v3VertexNormal, 0.0f);
    v3NormalOut = v4Normal.xyz;
}
```

29. Finally, we now need to add the corresponding inputs for the position and normal to the Fragment shader. Since the position was passed at location '0' and normal at location '1' care must be taken to ensure the inputs are correctly linked to the outputs.

```
layout(location = 0) in vec3 v3PositionIn;
layout(location = 1) in vec3 v3NormalIn;
```

30. Assuming everything went correctly with the moving of code then the new shader should be able to be used in place of the old. Try loading the new shader code instead of the old by modifying the initialise function and check that everything is working correctly.

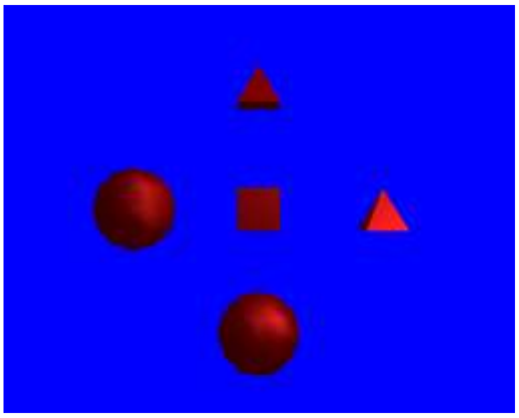
31. In order to compare the difference between using per-vertex and per-fragment shaders we will modify the code to allow both to be loaded at once. To do this we first need to add a second Program ID that can be used to store the second program. Then you need to duplicate the shader loading and linking code so that both shader programs are correctly loaded. For this tutorial the per-vertex program should be loaded into "`g_uiMainProgram`" and the per-fragment program into the second ID. Initially the per-vertex program should be set using "`glUseProgram`" during initialisation. Remember to make sure that all intermediate vertex/fragment shader IDs are cleaned up after linking and also add modified code to clean-up the additional program in "`GL_Quit`".

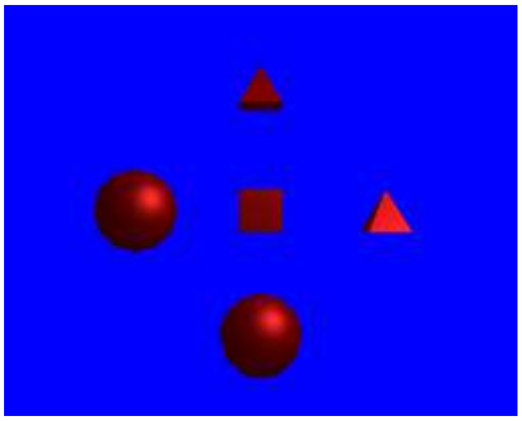
```
GLuint g_uiMainProgram;
GLuint g_uiSecondProgram;
```

32. The final step is to add code to the event loop so that we can dynamically change between different programs during runtime. We will do this by checking for an additional key input event. In this case we will use a key '1' button press ("`SDLK_1`") to change to using per-vertex lighting and a '2' key press ("`SDLK_2`") to change to per-fragment lighting. The appropriate code needs to be added to the event loop in the "`SDL_KEYDOWN`" section.

```
// Swap per-vertex and per-fragment shading
else if (Event.key.keysym.sym == SDLK_1)
    glUseProgram(g_uiMainProgram);
else if (Event.key.keysym.sym == SDLK_2)
    glUseProgram(g_uiSecondProgram);
```

33. You should now be able to compile and run your finished program. Test to make sure everything is working by pressing the '1' and '2' keys to toggle between using per-vertex and per-fragment lighting. Notice the different effects each lighting technique has especially on the specular highlight seen on the sphere. This should be particularly apparent while the sphere is moving.





Part 4: GGX Shading Function

Although the Blinn-Phong shader has been used extensively in computer graphics there are many modern shading algorithms that are capable of improved quality. One such shading technique is called the GGX shader. This algorithm is capable of representing a far greater range of materials by incorporating Fresnel and Micro-Facet geometry effects. This gives the GGX shader much greater visual quality and improves the ability for the model to accurately represent a much broader range of real-world materials. The GGX shader is also physically based which means that real-world measured data can be used as input which makes creating the material data much simpler. The GGX shader is also normalized by default and can be used in many advanced lighting models. The GGX shader can be described similarly to the Blinn-Phong shader. The difference is the specular component is now defined by a Fresnel reflectance term $F(\vec{\boldsymbol{l}},\vec{\boldsymbol{h}})$, Micro-Facet distribution term $D(\vec{\boldsymbol{h}})$ and a Visibility term $V(\vec{\boldsymbol{l}},\vec{\boldsymbol{v}},\vec{\boldsymbol{h}})$ such that:

$$L_V(\vec{\boldsymbol{v}}) = [\frac{1}{\pi}\rho_d(\overline{\hat{\vec{\boldsymbol{n}}}\cdot\hat{\vec{\boldsymbol{l}}}}) + F(\vec{\boldsymbol{l}},\vec{\boldsymbol{h}})D(\vec{\boldsymbol{h}})V(\vec{\boldsymbol{l}},\vec{\boldsymbol{v}},\vec{\boldsymbol{h}})](\overline{\hat{\vec{\boldsymbol{n}}}\cdot\hat{\vec{\boldsymbol{l}}}})E_L(\vec{\boldsymbol{l}})$$

For the GGX shader the Fresnel term is described by the Schlick Fresnel approximation. Using F_0 as the input specular coefficient the Schlick Fresnel approximation is defined as:

$$F(\vec{\boldsymbol{l}},\vec{\boldsymbol{h}}) = F_0 + (1 - F_0)(1 - (\vec{\boldsymbol{l}}\cdot\vec{\boldsymbol{h}}))^5$$

34. We will now add the GGX shader to our per-fragment shader code. We will do this in steps by adding functions for each of the components of the GGX model. The first component is the Fresnel term. We will create a function called "`schlickFresnel`" that is used to calculate the Schlick Fresnel approximation and return its result. Since we are using the specular coefficient as the value of F_0 then the return of the function must then be a "`vec3`".

```
vec3 schlickFresnel(in vec3 v3LightDirection, in vec3 v3Normal,
in vec3 v3SpecularColour)
{
    // Schlick Fresnel approximation
    float fLH = dot(v3LightDirection, v3Normal);
    return v3SpecularColour + (1.0f - v3SpecularColour) *
pow(1.0f - fLH, 5);
}
```

The Micro-Facet Distribution term used in the GGX shader is described by the Trowbridge-Reitz Normal Distribution Function (NDF). Using a roughness value r then the Trowbridge-Reitz function is defined as:

$$D(\vec{\boldsymbol{h}}) = \frac{r^2}{\pi\left[\overline{(\hat{\vec{\boldsymbol{n}}}\cdot\vec{\boldsymbol{h}})}^2(r^2 - 1) + 1\right]^2}$$

35. Next we must add a function to calculate the Normal Distribution Function. We will call this function “TRDistribution” and unlike the Fresnel function it will only return a “float”. The function will directly use the roughness value passed in from the input material data. It will also use the surface normal and half-vector passed in as input parameters.

```
float TRDistribution(in vec3 v3Normal, in vec3 v3HalfVector,
in float fRoughness)
{
    // Trowbridge-Reitz Distribution function
    float fNSq = fRoughness * fRoughness;
    float fNH = max(dot(v3Normal, v3HalfVector), 0.0f);
    float fDenom = fNH * fNH * (fNSq - 1.0f) + 1.0f;
    return fNSq / (M_PI * fDenom * fDenom);
}
```

The final component in the GGX shader is the Visibility function. The GGX Visibility function is actually a combination of the GGX Geometry function and the cosine weighting factor found in Micro-Facet models. This function is normally described directly as the Geometry function alone. However, by combining it with the cosine factor the overall equation can be simplified due to the cancelation of terms, as a result it is much more desirable to combine these into the Visibility function. This function uses the same roughness r value as the Normal Distribution Function. The GGX Visibility function is defined as:

$$V(\vec{\boldsymbol{l}},\vec{\boldsymbol{v}},\vec{\boldsymbol{h}}) = \frac{\mathbf{1}}{\overline{(\hat{\vec{\boldsymbol{n}}}\cdot\hat{\vec{\boldsymbol{l}}})} + \sqrt{\boldsymbol{r}^2 + (\mathbf{1}-\boldsymbol{r}^2)\overline{(\hat{\vec{\boldsymbol{n}}}\cdot\hat{\vec{\boldsymbol{l}}})}^2}}\,\frac{\mathbf{1}}{\overline{(\hat{\vec{\boldsymbol{n}}}\cdot\hat{\vec{\boldsymbol{v}}})} + \sqrt{\boldsymbol{r}^2 + (\mathbf{1}-\boldsymbol{r}^2)\overline{(\hat{\vec{\boldsymbol{n}}}\cdot\hat{\vec{\boldsymbol{v}}})}^2}}$$

36. We must now add the Visibility function to our Fragment shader. We will call this function “GGXVisibility” and just like the NDF function it will only return a “float”.

```
float GGXVisibility(in vec3 v3Normal, in vec3 v3LightDirection,
in vec3 v3ViewDirection, in float fRoughness)
{
    // GGX Visibility function
    float fNL = max(dot(v3Normal, v3LightDirection), 0.0f);
    float fNV = max(dot(v3Normal, v3ViewDirection), 0.0f);
    float fRSq = fRoughness * fRoughness;
    float fRMod = 1.0f - fRSq;
    float fRecipG1 = fNL + sqrt(fRSq + (fRMod * fNL * fNL));
    float fRecipG2 = fNV + sqrt(fRSq + (fRMod * fNV * fNV));

    return 1.0f / (fRecipG1 * fRecipG2);
}
```

37. The final step is to add the complete GGX function. This function should call “`schlickFresnel`”, “`TRDistribution`” and “`GGXVisibility`” according to the GGX shader equation. We will create this function and call it “GGX”. Just like the Blinn-Phong function it will return the final colour in a “`vec3`”.

```
vec3 GGX(in vec3 v3Normal, in vec3 v3LightDirection, in vec3 v3ViewDirection,
in vec3 v3LightIrradiance, in vec3 v3DiffuseColour, in vec3 v3SpecularColour,
in float fRoughness)
{
    // Calculate diffuse component
    vec3 v3Diffuse = v3DiffuseColour * M_RCPPI;

    // Calculate half vector
    vec3 v3HalfVector = normalize(v3ViewDirection + v3LightDirection);

    // Calculate Toorance-Sparrow components
    vec3 v3F = schlickFresnel( v3LightDirection, v3HalfVector,
v3SpecularColour );
    float fD = TRDistribution(v3Normal, v3HalfVector, fRoughness);
    float fV = GGXVisibility(v3Normal, v3LightDirection, v3ViewDirection,
fRoughness);

    // Combine diffuse and specular
    vec3 v3RetColour = v3Diffuse + (v3F * fD * fV);

    // Multiply by view angle
    v3RetColour *= max(dot(v3Normal, v3LightDirection), 0.0f);

    // Combine with incoming light value
    v3RetColour *= v3LightIrradiance;

    return v3RetColour;
}
```

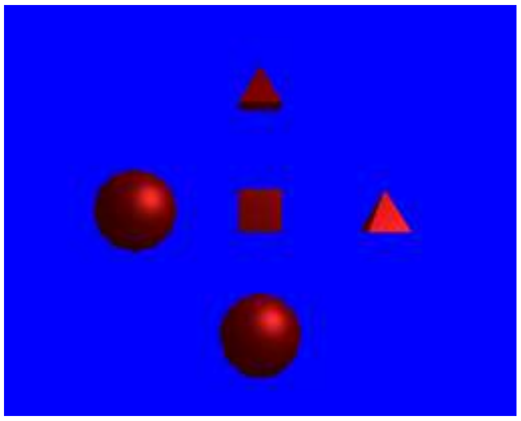
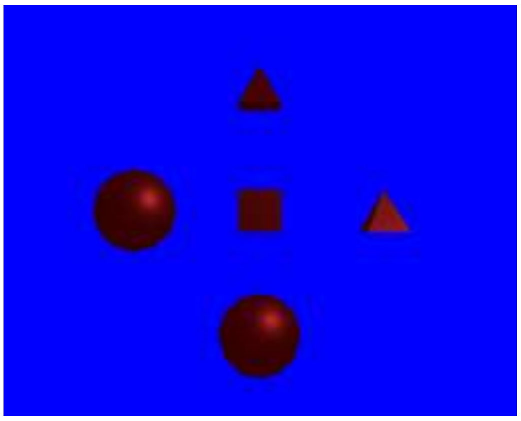
38. To use the GGX shader the call to “`blinnPhong`” in the fragment shaders main function needs to be replaced with a call to “GGX” instead.

39. The GGX shader uses a roughness value just like the normalized Blinn-Phong. However, they are not equivalent. For instance, Phong roughness values often range from 20-100 while GGX roughness values are physically based and range from 0-1. To use the GGX shader we need to update the host material data’s roughness value to one that will work with the GGX shader. An equivalent value to the existing Phong roughness term is 0.35.

40. In order to continue to use the Phong shader with the same input roughness value we need to add additional code to the Blinn-Phong shader to convert the GGX roughness value to one that can be used in a Phong shader (remember to update the per-vertex shader as well). The equation to convert from GGX roughness to Phong roughness is $n = \frac{2}{r^2} - 2$.

```
// Convert roughness to Phong shininess
float fRoughnessPhong = (2.0f / (fRoughness * fRoughness)) - 2.0f;
```

41. You should now be able to run your code and test the GGX shader.

Part 5: Shader Subroutines

Often many real-world programs need to change shaders in order to perform different calculations. This can be rather inefficient as often many programs share the same code and only a small portion needs changing. OpenGL 4.0 and onwards provides the concept of subroutines in order to overcome this issue. An OpenGL subroutine is a function that can be swapped at runtime for a different function with the same interface. This way certain components of a shader program can be changed at runtime without needing multiple shader programs.

42. In this tutorial we want to be able to easily swap between the Blinn-Phong and the GGX shader in the per-fragment program. To do this we will use OpenGL subroutines. To use a subroutine, the first step is to declare the interface that is shared by all subroutines of that type. Here we want to use a different function to perform shading operations where each function uses a different shading equation (BRDF). To do this we will declare a subroutine in our per-fragment fragment shader using the GLSL keyword “`subroutine`”. We will call this subroutine “BRDF” and declare its interface as one that takes 6 “`vec3`”’s and a “`float`” as input and return a single “`vec3`” as output.

```
subroutine vec3 BRDF(in vec3, in vec3, in vec3, in vec3, in vec3, in vec3,
in float);
```

43. We now need to change the definition of each of our shading functions “`blinnPhong`” and “GGX” to specify that they are now subroutines with the same interface as the declared subroutine “BRDF”. This declares that both functions are interchangeable and can be referenced by the subroutine name “BRDF”. We use “`layout(index)`” to explicitly specify an ID for each subroutine that can later be used to select between them. Here the Blinn-Phong shader is given an index of ‘0’ while the GGX shader should be given an index of ‘1’.

```
layout(index = 0) subroutine(BRDF)
vec3 blinnPhong( ...
```

44. Now that we have 2 functions that are usable as subroutines we now need to add a uniform to the shader code that can be used to pass the current index of the subroutine that should actually be used during code execution. This requires adding a new uniform to the code that has the “`subroutine`” type.

```
layout(location = 0) subroutine
uniform BRDF BRDFUniform;
```

This uniform specifies the subroutine that it applies to (in this case "BRDF") and then gives a name for the uniform that can be used to access it from external code. We will give the uniform a name of "BRDFUniform" and explicitly set its location to '0'.

45. In order to call the subroutine, we now use the name "BRDFUniform" to call the currently bound subroutine function. To do this we must change the existing call to the GGX function "GGX" in the main function with a call to the uniform name "BRDFUniform".

```
// Perform shading
v3RetColour = BRDFUniform( ...
```

46. Now that we have setup and initialised the subroutines we will add some extra code to allow for changing the subroutines on the fly during program execution. Just like with code for changing programs we added previously we will add extra event handling for key presses. Now instead of a key '2' press setting the second program we now want the same key press to set the second program and its first subroutine. We then add code so that a key '3' press ("SDLK_3") uses the second program but with the second subroutine. Since it is possible for the program flow to jump from key press '1' to key press '3' it cannot be assumed that a particular program is already loaded. As such we must explicitly load the required program each key press. This will allow us to press '2' to use per-fragment Blinn-Phong shading and then press '3' to swap to per-fragment GGX shading. All while still allowing for a key press of '1' to use per-vertex lighting. Changing subroutines is done with a call to the "glUniformSubroutinesuiv" function. This function sets all available subroutines in a shader stage at once. The first input is therefore the stage to set (in this case we are still working on the fragment shader). The second input is the number of subroutines being set which must equal the number of subroutines found in the shader stage (in this case there is only 1). The final parameter is an array of IDs for each subroutine in the shader stage where the subroutine at element '0' in the array corresponds to the subroutine uniform with ID '0' (and so on for all subroutines). As we have only 1 subroutine that we assume has ID of '0' we pass a single value corresponding to the first subroutine (blinnPhong) passed using index '0' of a single element array.

```
// Swap subroutines
else if (Event.key.keysym.sym == SDLK_2) {
    glUseProgram(g_uiSecondProgram);
    GLuint g_uiSubRoutines[] = {0};
    glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1,
 &g_uiSubRoutines[0]);
} else if (Event.key.keysym.sym == SDLK_3) {
    glUseProgram(g_uiSecondProgram);
    GLuint g_uiSubRoutines[] = {1};
    glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1,
 &g_uiSubRoutines[0]);
}
```

47. You should now be able to compile and run the completed program. Test each of the key inputs '1', '2' and '3' to make sure each shader stage is working correctly and to compare the outputs.

Part 6: Extra

48. In this tutorial we only have 1 single material. Update the program so that each object has its own material each with its own different material properties (you can make up any that you wish).

49. In order to allow for more control over lighting falloff an extended falloff function needs to be used. Update the light falloff function to use the following extended falloff function:

$$f_{dist}(r) = \frac{1}{k_{r0} + k_{r1}r + k_{r2}r^2}$$

50. Add 2 more additional lights to the program. Instead of creating additional light UBOs multiple lights can be added to the existing UBO by making it store an array of light data. The input to the shader can then be modified to pass multiple lights. Since GPU resources need to be allocated for the light array it needs to have a fixed pre-allocated size. Here we will use “MAX_LIGHTS” to specify the maximum size of the light array. We then add a uniform “iNumPointLights” that is used to specify the actual number of lights in the array (must be less than “MAX_LIGHTS”). The shader code itself then needs to be updated with a loop that goes over each light and adds its contribution to the final output (remember to update both programs). The number of lights can then be set for each program by using “glProgramUniform1i” which accepts 3 parameters; the first being the shader program to set the uniform in, the index of the uniform to set (in this case ‘0’) and then the value to set it to (in this case ‘3’). This needs to be done for both programs.

```
#define MAX_LIGHTS 16
layout(binding = 2) uniform PointLightData {
    PointLight PointLights[MAX_LIGHTS];
};
layout(location = 0) uniform int iNumPointLights;
```

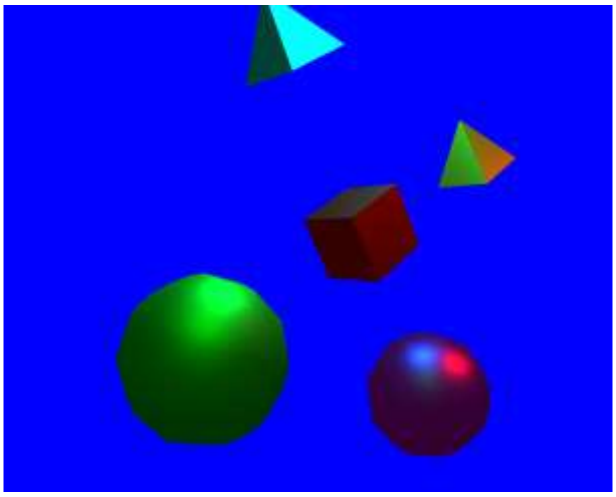
```
// Set number of lights
glProgramUniform1i(g_uiMainProgram, 0, 3);
```

In the host code first change the existing light so that it outputs red light with intensity of 1 (i.e. (1.0,0.0,0.0)). Then create a second light at position (-6.0, 6.0, 6.0) and set it to output green light with intensity 1. Finally set the third light to a position of (0.0, 6.0, 0.0) and set its output to cyan light with intensity of 1.

51. Notice how areas under direct light seem saturated and overly bright. This is due to the diffuse term not being scaled with respect to the specular term. To fix this add a Fresnel modifier to the diffuse term to scale it accordingly. Add this modifier to the GGX shader in order to see its effect.

$$f_d(\vec{\boldsymbol{l}},\vec{\boldsymbol{v}}) = f_d(\vec{\boldsymbol{l}},\vec{\boldsymbol{v}})\left(1 - F(\vec{\boldsymbol{l}},\vec{\boldsymbol{n}})\right)$$

Note: The existing value for F calculated using the half angle can be used in place of calculating with $\vec{\boldsymbol{n}}$. This introduces minimal loss in visual quality but improves performance.



Note: Using with older OpenGL versions

This tutorial uses OpenGL functionality that is not found in older versions. For instance, shader subroutine indexes are explicitly specified in shader code using "`layout(index)`" and the subroutine uniform is explicitly specified using "`layout(location)`" which is a feature only found in OpenGL 4.3 and beyond. Similar code can still be used on older versions (subroutines are available since version 4.0) by allowing OpenGL to create the subroutine indexes and then manually probing them.

To use a subroutine uniform on older OpenGL versions "`layout(location)`" can be replaced by getting the location of the uniform in host code. This can be achieved using a call to "`glGetSubroutineUniformLocation`". This is used to pass the programs shader stage that you wish to retrieve the uniform from and the name of the subroutine uniform you want to get. In this tutorial the subroutine was in the fragment shader "`GL_FRAGMENT_SHADER`" and was called "`BRDFUniform`" so the following additional host code would be needed:

```
GLint iSubRoutineU = glGetSubroutineUniformLocation(g_uiSecondProgram,
GL_FRAGMENT_SHADER, "BRDFUniform");
```

Next you need to get the IDs of each specific subroutine that can be bound to the subroutine uniform. This is done using the OpenGL function "`glGetSubroutineIndex`" which takes as input the shader program ID, the programs shader stage to check and the name of the subroutine to return. In this tutorial there are 2 subroutines so you need to get the indexes for the "`blinnPhong`" and "`GGX`" subroutines.

```
// Get available subroutines
GLuint g_uiSubRoutines[2];
g_uiSubRoutines[0] = glGetSubroutineIndex(g_uiSecondProgram,
GL_FRAGMENT_SHADER, "blinnPhong");
g_uiSubRoutines[1] = glGetSubroutineIndex(g_uiSecondProgram,
GL_FRAGMENT_SHADER, "GGX");
```

The retrieved subroutine indexes can then be used as normal to swap between the subroutines by using "`glUniformSubroutinesuiv`" and passing "`g_uiSubRoutines[0]`" for the Blinn-Phong subroutine or "`g_uiSubRoutines[1]`" for GGX.

This tutorial also used explicit uniform locations which requires OpenGL 4.3 or above. These can be converted to older OpenGL versions by removing the use of "`layout(location)`" and adding in extra host code to retrieve the OpenGL location for the uniform and using that. This can be achieved using a call to "`glGetUniformLocation`" and passing in the shader program to retrieve the index from as well as the name of the uniform. This tutorial has a single uniform that was used to store the number of lights. Using "`glGetUniformLocation`" the OpenGL location of the uniform can be retrieved and that can be used as the second parameter of "`glProgramUniform1i`" in order to set its value (in this tutorial it was set to '3'). Note: This must be done for every program individually.

```
// Get light uniform location and set
GLuint uiUIndex = glGetUniformLocation(g_uiMainProgram, "iNumPointLights");
glProgramUniform1i(g_uiMainProgram, uiUIndex, 3);
```

Using "`glGetUniformLocation`" requires OpenGL 4.1. Older versions of OpenGL can be supported by using "`glUniform1i`" instead. This function is similar to "`glProgramUniform1i`" except that it doesn't take the shader program as input as it instead only works on the currently bound shader program.

```
// Get light uniform location and set
GLuint uiUIndex = glGetUniformLocation(g_uiMainProgram, "iNumPointLights");
glUseProgram(g_uiMainProgram);
glUniform1i(uiUIndex, 3);
```

Explicit program linkage was also used in this tutorial to supply a location to outputs from a shader stage and then provide a matching location for the corresponding inputs of the next shader stage. This requires OpenGL 4.1 or above but can be used on any older version by removing the "`layout(location)`" qualifier entirely. Without this qualifier OpenGL will try to dynamically match the interface variables by using the order in which they are declared in and the variable type to detect a match. OpenGL will also use the name of the variable to match an input with an output that has the same name. The interface variables in this tutorial have different names but providing they are declared in the same order for outputs and inputs then OpenGL will match them correctly without needing "`layout(location)`".