Lab 8 – Bumpmapping

In this exercise we are going to implement Bumpmapping. This means that we will have to adapt our application code in order to interface with this new set of shaders.

1. Textures and Lights

The first task we are going to accomplish is add in textures to our shaders. This require that we move some of the calculations from the Vertex to Fragment shader.

1.1 Vertex Changes

The first thing to check is that your Vertex has texture coordinates defined. Open up **Vertex.h** and ensure you have the following Vertex Structure defined

```
struct Vertex
{
	vec3 position;
	vec3 normal;
	vec2 texCoords;
};
```

You may have to add a using statement for **glm::vec2** at the top of the file.

This change will mean that you will have to update the **glBindAttributeLocation** in **Material.cpp**(the texture coordinates should be location 2 with the name "**vertexTexCoords**"). You will also need to correctly layout the vertices using **glVertexAttribPointer** in **Mesh.cpp**. If you are unsure of how to do this, please check the existing code or ask the lab demonstrator.

1.2. Material – Diffuse Texture

We are going to be adding Textures into the shader we need to think from a design point of view where these textures should be held in the Application Stage. A perfect place for these textures would the **Material** class, this class already deals with how an object looks(colours and shaders) so it makes sense to use the material to store all textures.

Open up **Material.h** and add a **GLuint** called **m_DiffuseMap** to the **private** section of the class. You should also declare the following functions

- **loadDiffuseMap** which returns **void** and takes in **const std::string& filename** as a parameter
- **getDiffuseMap** which returns a **GLuint** and takes in no parameters(empty brackets)**.**

Now switch to **Material.cpp** and set **m_DiffuseMap** to **0** and in the **destroy** function use the **glDeleteTextures** function to delete the texture(the 1st parameter should be **1** and the 2nd parameter should be **&m_DiffuseMap**)

We now ready to implement the **loadDiffuseMap** and **getDiffuseMap** functions, but first we will have to add an **include** statement for the "**Texture.h**" file so that we can use our texture loading functions. The **getDiffuseMap** will simply return **m_DiffuseMap** variable and the **loadDiffuseMap** will call the **loadTextureFromFile** function using the **filename** as a parameter and the result of this function will be assigned to **m_DiffuseMap.**

Finally we need to bind the texture, navigate to the **bind** function in **Material.cpp** and in the function called **glActiveTexture** with **GL_TEXTURE0** as the parameter. We can now bind the texture, called **glBindTexture** with **GL_TEXTURE_2D** and **m_DiffuseMap** as the parameters. This will bind the **m_DiffuseMap** to Texture unit **0**(GL_TEXTURE**0**) and any additional textures will be bound to the subsquent Texture Units(GL_TEXTURE1,GL_TEXTURE2 etc etc)

At this point you should build the application, if you have any errors please contact the lab demonstrator.

1.3 Shader Changes

We are going to make some changes to our previous light shader, copy the **specularVS.glsl** & **specularFS.glsl** and rename the files to **directionalLightTextureVS.glsl** & **directionLightTextureFS.glsl**. Add these new files to Visual Studio.

Open up **directionalLightTextureVS.glsl** and ensure you have the following Global Values

```
in vec3 vertexPosition;
in vec3 vertexNormals;
in vec2 vertexTexCoords;

out vec3 vertexNormalOut;
out vec3 cameraDirectionOut;
out vec3 lightDirectionOut;
out vec2 texCoordsOut;

uniform mat4 MVP;
uniform mat4 Model;

uniform vec3 lightDirection;
uniform vec3 cameraPosition;
```

You should remove all other Global Values!

The main function will change a little bit, it will pass some of globals(Texture Coordinates) but will also carry out some Transformations on others(Normals to World Space)

```
void main()
{

	vec3 vertexNormalModel = normalize(Model*vec4(vertexNormals, 0.0f)).xyz;
	vec3 worldPos = (Model*vec4(vertexPosition, 1.0)).xyz;
	vec3 cameraDir = normalize(cameraPosition - worldPos);

	lightDirectionOut = normalize(lightDirection);
	vertexNormalOut = normalize(vertexNormalModel);
	cameraDirectionOut = normalize(cameraDir);

	texCoordsOut = vertexTexCoords;
	gl_Position = MVP * vec4(vertexPosition, 1.0);
}
```

The above function simply calculates some of the values needed for the reflectance calculation and then sets the corresponding output parameters. One thing to note that we could eliminate 3 instructions and storage for 3 parameters by directly setting the output values to the results of the calculation but we have created this shader as a basis for other effects such as Bumpmapping.

Switch to **directionLightTextureFS.glsl** and ensure that the shader has the following global values

```
out vec4 FragColor;

in vec3 vertexNormalOut;
in vec3 cameraDirectionOut;
in vec3 lightDirectionOut;

in vec2 texCoordsOut;

uniform vec4 ambientMaterialColour;
uniform vec4 diffuseMaterialColour;
uniform vec4 specularMaterialColour;
uniform float specularPower;

uniform vec4 ambientLightColour;
uniform vec4 diffuseLightColour;
uniform vec4 specularLightColour;

uniform sampler2D diffuseMap;
```

Now the main function should look like the following

```
void main()
{

        float diffuseTerm = dot(vertexNormalOut, lightDirectionOut);
        vec3 halfWayVec = normalize(cameraDirectionOut + lightDirectionOut);
        float specularTerm = pow(dot(vertexNormalOut, halfWayVec),
                                                        specularPower);

        vec4 diffuseTextureColour = texture(diffuseMap, texCoordsOut);

        FragColor = (ambientMaterialColour*ambientLightColour) +
((diffuseMaterialColour + diffuseTextureColour)*diffuseLightColour*diffuseTerm) +
                (specularMaterialColour *specularLightColour*specularTerm);
}
```

A lot of this code should be familiar from the last exercise, we have simply moved the reflectance calculation to the Fragment Shader. The major difference is that we carry out a texture lookup and add the result of the lookup the material colour.

1.4 Using the Shaders

The rest of this exercises assumes you have already have a working example from the Reflectance Model Lab.

Add/change the **init** function to load a model and use the above shaders.

```
std::string modelPath = ASSET_PATH + MODEL_PATH + "armoredrecon.fbx";
GameObject * go = loadFBXFromFile(modelPath);
for (int i = 0; i < go->getChildCount(); i++)
{
        Material * material = new Material();
        material->init();
        std::string vsPath = ASSET_PATH + SHADER_PATH +
                                                "/directionalLightTextureVS.glsl";
        std::string fsPath = ASSET_PATH + SHADER_PATH +
                                                "/directionalLightTextureFS.glsl ";
        material->loadShader(vsPath, fsPath);

        std::string diffTexturePath = ASSET_PATH + TEXTURE_PATH +
                                                        "/armoredrecon_diff.png";
        material->loadDiffuseMap(diffTexturePath);
        go->getChild(i)->setMaterial(material);
}
go->getTransform()->setPosition(2.0f, -2.0f, -6.0f);
displayList.push_back(go);
```

The code above is fairly similar to previous labs, the only real difference is the loading of the diffuse texture. This **armoredrecon_diff.png** file can be downloaded from GCU Learn.

Now move down to the **renderGameObject** function and inside the **if statement** ensure that you are retrieving the following variables

```
GLint MVPLocation = currentMaterial->getUniformLocation("MVP");
GLint ModelLocation = currentMaterial->getUniformLocation("Model");
GLint ambientMatLocation =
        currentMaterial->getUniformLocation("ambientMaterialColour");
GLint ambientLightLocation =
        currentMaterial->getUniformLocation("ambientLightColour");
GLint diffuseMatLocation =
        currentMaterial->getUniformLocation("diffuseMaterialColour");
GLint diffuseLightLocation =
        currentMaterial->getUniformLocation("diffuseLightColour");
GLint lightDirectionLocation =
        currentMaterial->getUniformLocation("lightDirection");
GLint specularMatLocation =
        currentMaterial->getUniformLocation("specularMaterialColour");
GLint specularLightLocation =
        currentMaterial->getUniformLocation("specularLightColour");
GLint specularpowerLocation =
        currentMaterial->getUniformLocation("specularPower");
GLint cameraPositionLocation =
        currentMaterial->getUniformLocation("cameraPosition");
GLint diffuseTextureLocation =
        currentMaterial->getUniformLocation("diffuseMap");
```

You can double the above with the uniforms contained in the Vertex and Fragment shader

Now we have retrieved the locations we are now able to send the variables, again this assumes you have completed the previous lab.

```
glUniformMatrix4fv(ModelLocation, 1, GL_FALSE, glm::value_ptr(Model));
glUniformMatrix4fv(MVPLocation, 1, GL_FALSE, glm::value_ptr(MVP));
glUniform3fv(cameraPositionLocation, 1, glm::value_ptr(cameraPosition));

glUniform4fv(ambientMatLocation, 1, glm::value_ptr(ambientMaterialColour));
glUniform4fv(diffuseMatLocation, 1, glm::value_ptr(diffuseMaterialColour));
glUniform4fv(specularMatLocation, 1, glm::value_ptr(specularMaterialColour));
glUniform1f(specularpowerLocation, specularPower);

glUniform4fv(ambientLightLocation, 1, glm::value_ptr(ambientLightColour));
glUniform4fv(diffuseLightLocation, 1, glm::value_ptr(diffuseLightColour));
glUniform4fv(specularLightLocation, 1, glm::value_ptr(specularLightColour));
glUniform3fv(lightDirectionLocation, 1, glm::value_ptr(lightDirection));

glUniform1i(diffuseTextureLocation, 0);
```

Again this should be very familiar to previous labs, you should also check to see if the **glUniform*** lines match up to the **getUniformLocation.**

If you build and run the application, you should see a lit textured armoured car. You may find that the model is a bit too bright, you may want to change the diffuse material colour to black(0.0f,0.0f,0.0f,1.0f).

1.5 Material – Specular Texture

We now have support for diffuse textures, we should add support for Specular Textures. This will be very similar to the Diffuse Texture, you should follow the steps in **1.2** to add this Specular Texture support to the **Material** class. Remember to bind the Texture to Texture unit 1(**GL_TEXTURE1**)!

1.6 Updating Shader

We need to update the fragment shader to support the Specular Textures, open up Fragment shader and add a **uniform** of type **sampler2D** called **specMap**.

Move down to the **main** function and add the line to lookup the texture, this should be exactly the same as the lookup for the Diffuse Map. Again like the diffuse map you set the result of the lookup to a **vec3** called **specTextureColour**.

Now we have this colour, we can then use it in the reflectance calculation, by adding the **specTextureColour** to the **specularMaterialColour**. This is exactly the same as diffuse.

1.7 Using the Shader – update

We now need implement the Application side of things. We need to add code to load the Specular Map("armoredrecon_spec.png", which can be downloaded from GCU Learn), retrieve the uniform location and send the texture. This is

almost exactly the same as the diffuse texture, so you should be able to replicate this for the specular texture(see step 1.4).

At this point you should be able to build and run the application, you should see an armoured car which is lit by a specular light. If the specular highlight is too bright then reduce the specular material to black.

2. Bump mapping

In this part of the exercise we are going to implement Bumpmapping, this will involve changing out Vertices to contain binormals and tangent normals(used to calculate a Tangent Matrix), our model loading code will be changed to read-in or calculate the new components of a vertex, we will implement our Bumpmapping shader, make changes to the Material class to support the loading of the Normal Map and finally change our Application code to support the shader.

2.1 Bumpmapping Vertex Shader

Copy the **directionalLightTextureVS.glsl** and paste a copy of the file into the shader folder. Rename this shader as **bumpMappingVS.glsl**.

Ensure that the shader as the following input parameters

```
in vec3 vertexPosition;
in vec3 vertexNormals;
in vec2 vertexTexCoords;
in vec3 vertexTangents;
in vec3 vertexBinormals;
```

Inside the main function, we want to calculate a tangent matrix. This matrix will be used to transform all vectors to tangent space which will enable us to carry out all lighting calculations in the same space. Add the following line to the top of the **main** function to calculate the matrix

```
mat3 tangentMatrix = mat3(normalize(vertexNormals), normalize(vertexTangents),
normalize(vertexBinormals));
```

The above line constructs a 3x3 matrix using the vertex normals, tangent normals and the binormals. This can be used to transform any vector into tangent space(aka Texture Space).

Still in the main function you should replace the lines which set the output values for the normals, light direction and camera direction with the following

```
lightDirectionOut = normalize(tangentMatrix*lightDirection);
cameraDirectionOut = normalize(tangentMatrix*cameraDir);
```

The above lines simply transform the camera and light direction into tangent space, we also normalize the result at the same time so we can still use the simplified version of reflectance calculation. Another point to note is that we no longer need to out the normals as we are fetching these from the Bump Map, this means you can remove any references to the **vertexNormalOut** from the shader.

2.2 Bumpmapping Vertex Shader

Copy the **directionalLightTextureFS.glsl** and paste a copy of the file into the shader folder. Rename this shader as **bumpMappingFS.glsl**.

The only real change to Fragment shader is that we take in a new **uniform sampler2D** called **bumpMap**, in the main function we fetch the normals from the bumpmap and rescale them to -1 to 1 range and finally we use these normals instead of the vertex normals.

Add the following global uniform to the shader, this will hold the bumpmap

```
uniform sampler2D bumpMap;
```

Now inside the main function, we are going to carry out a texture lookup to retrieve the normals and then remap this normal to -1 to 1 range(while normalizing). Add the following to the top of the **main** function

```
vec3 bumpNormals = normalize(2.0 * texture2D(bumpMap, texCoordsOut).rgb - 1.0);
```

Finally replace any mention of **vertexNormalOut** in the **main** function with the above **bumpNormals**. At this point you can also delete the global **in vec3 vertexNormalOut.**

2.3 Vertex Changes

You should notice from the shader that we have cnaged the input vertices. You should changes the Vertices on the Application side to match this new vertex layout.

Open up **Vertex.h** and ensure that the vertex structure matches the code below.

```
struct Vertex
{
	vec3 position;
	vec3 normal;
	vec2 texCoords;
	vec4 colours;
	vec3 tangentNormals;
	vec3 binormals;
};
```

This change will mean that you will have to update the **glBindAttributeLocation** in **Material.cpp**, you can use the input values of the shader as a reference. You will also need to correctly layout the vertices using **glVertexAttribPointer** in **Mesh.cpp**. If you are unsure of how to do this, please check the existing code or ask the lab demonstrator.

2.4. Model loading Changes

Since we are using Bumpmapping we need to load in the Tangent and bi-normals, these can usually be loaded from the FBX file but some modeling packages don't export these by default so it is easier to calculate.

Download **FBXLoader.zip** from GCU Learn and replace the existing **FBXLoader.cpp** and **h** with the files in the zip.

Build the application at this point to ensure there are no errors!

2.5 Material – Bumpmap Texture

Follow the steps **1.2** to add support for loading and storing a Bumpmap texture. Remember to use Texture unit 2(**GL_TEXTURE2**) in the bind function.

2.6 Using the Shader

See steps **1.4**, we need to load the Bumpmap texture("armoredrecon_N.png" again on GCU Learn), retrieve and send the texture.

Build and run the application, you should see a 'Bumpy version of the vehicle which should look like the following image


Lab 6