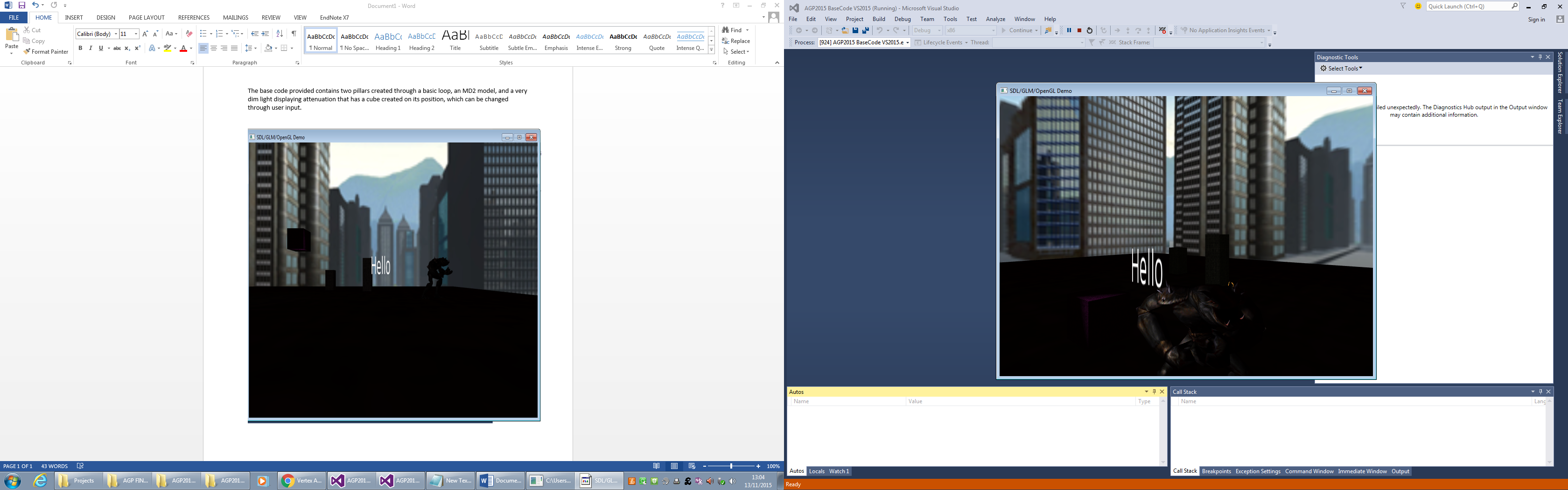
**AGP 2015 – Multiple Lighting Tutorial**

**B00250737**

The base code provided contains two pillars created through a basic loop, an MD2 model, and a very dim light displaying attenuation that has a cube created on its position, which can be changed through user input:



The lightStruct and the setLight function are very similar to the ordinary rt3d library, with the only changes being the addition of attenuation variables, which is naturally required if you wish for the different lights you have to have different attenuation effects.

However, LightPos and lightPosition variables throughout the program are still set up for use of a single light, which is inconvenient as we need our multiple light positions to be used in the calculations for the lighting to be correct.

In this tutorial’s example, we will be using the phong-tex fragment and vertice shaders.

The first step will be editing ‘uniform vec4 lightPosition’ in the vertice shader so that it is an array capable of holding the multiple positions we will soon be sending in:

‘uniform vec4 lightPosition[3]’

*Due to this, the ex\_D and ex\_L variables will require the same treatment in order to store the different results from resultant calculations:*

out vec3 ex\_L[3];

out float ex\_D[3];

*With arrays ready to store information, the calculation itself must now be altered slightly. A run through of the entire calculation will be needed for each light, so set it all up within a standard ‘for’ loop with the ‘lightPosition’, ‘ex\_D’, and ‘ex\_L’ parts modified so that each run through uses a new position in their arrays:*

for(int i = 0; i < 3; i++)

{

// vertex into eye coordinates

vec4 vertexPosition = modelview \* vec4(in\_Position,1.0);

ex\_D[i] = distance(vertexPosition,lightPosition[i]);

// Find V - in eye coordinates, eye is at (0,0,0)

ex\_V = normalize(-vertexPosition).xyz;

// surface normal in eye coordinates

// taking the rotation part of the modelview matrix to generate the normal matrix

// (if scaling is includes, should use transpose inverse modelview matrix!)

// this is somewhat wasteful in compute time and should really be part of the cpu program,

// giving an additional uniform input

mat3 normalmatrix = transpose(inverse(mat3(modelview)));

ex\_N = normalize(normalmatrix \* in\_Normal);

// L - to light source from vertex

ex\_L[i] = normalize(lightPosition[i].xyz - vertexPosition.xyz);

ex\_TexCoord = in\_TexCoord;

gl\_Position = projection \* vertexPosition;

}

*Naturally, the fragment shader will also need to be changed to accommodate the new data being sent in:*

in float ex\_D[3];

…

in float ex\_D[3];

*Multiple run-throughs of the code in the fragment shader will also be required. A neat way of going about this is putting all the code in main (replacing the ‘out\_Color’ line with a line simply returning the ‘litColour’ variable at the end) in its own function that can be called multiple times from main:*

vec4 PointLightCalculation(int number)

{

// Ambient intensity

vec4 ambientI = light[number].ambient \* material.ambient;

// Diffuse intensity

vec4 diffuseI = light[number].diffuse \* material.diffuse;

diffuseI = diffuseI \* max(dot(normalize(ex\_N),normalize(ex\_L[number])),0);

// Specular intensity

// Calculate R - reflection of light

vec3 R = normalize(reflect(normalize(-ex\_L[number]),normalize(ex\_N)));

vec4 specularI = light[number].specular \* material.specular;

specularI = specularI \* pow(max(dot(R,ex\_V),0), material.shininess);

float attenuation = ((light[number].attConst + light[number].attLinear \* ex\_D[number] + light[number].attQuadratic \* (ex\_D[number]\*ex\_D[number])));

vec4 tmp\_Color = (((diffuseI + specularI + ambientI)/attenuation));

vec4 litColour = vec4(tmp\_Color.rgb, 1.0)\* texture(textureUnit0, ex\_TexCoord);

return litColour;

}

*(Remember that the function will need to be defined before main!)*

*The ‘int number’ parameter will be used similarly to the previous ‘for’ loop, taking in a new value with each run. In this example, the loop will be controlled by a variable that we will define just after the structs, we will use this variable to store how many lights to create of the lightStruct like so:*

#define LIGHT\_COUNT 3

uniform lightStruct light[LIGHT\_COUNT];

*And now to alter the main function:*

void main(void)

{

vec4 result;

for(int i = 0; i < (LIGHT\_COUNT); i++)

{

result = (result + PointLightCalculation(i));

}

out\_Color = result;

}

*Our shaders are now fit for multiple lights, changing the setLight function in the rt3d library so that it can be used for multiple lights in a good idea. Go to the rt3d.cpp file.*

*Adding a new parameter ‘GLuint lightNumber’ to the setLight function will allow us to use arrays in the ‘glGetUniformLocation’ functions, which is handy as it allows us to use the setLight function for all the point lights we could need.*

*However, this requires a bit of setting up, first lightNumber must be converted to string, after which the uniform location calls require editing to include the arrays in their search. The updated function should therefore look like this:*

void setLight(const GLuint program, const lightStruct light, GLuint lightNumber)

{

string number = std::to\_string(lightNumber);

// pass in light data to shader

int uniformIndex = glGetUniformLocation(program, ("light[" + number + "].ambient").c\_str());

cout << "Amb" << lightNumber << ": " << uniformIndex << endl;

glUniform4fv(uniformIndex, 1, light.ambient);

uniformIndex = glGetUniformLocation(program, ("light[" + number + "].diffuse").c\_str());

glUniform4fv(uniformIndex, 1, light.diffuse);

cout << "Diffuse" << lightNumber << ": " << uniformIndex << endl;

uniformIndex = glGetUniformLocation(program, ("light[" + number + "].specular").c\_str());

glUniform4fv(uniformIndex, 1, light.specular);

cout << "Spec" << lightNumber << ": " << uniformIndex << endl;

uniformIndex = glGetUniformLocation(program, ("lightPosition[" + number + "]").c\_str());

glUniform4fv(uniformIndex, 1, light.position);

cout << "Position" << lightNumber << ": " << uniformIndex << endl;

uniformIndex = glGetUniformLocation(program, ("light[" + number + "].attConst").c\_str());

glUniform1f(uniformIndex, light.attConst);

cout << "Constant" << lightNumber << ": " << uniformIndex << endl;

uniformIndex = glGetUniformLocation(program, ("light[" + number + "].attLinear").c\_str());

glUniform1f(uniformIndex, light.attLinear);

cout << "Linear" << lightNumber << ": " << uniformIndex << endl;

uniformIndex = glGetUniformLocation(program, ("light[" + number + "].attQuadratic").c\_str());

glUniform1f(uniformIndex, light.attQuadratic);

cout << "Quadratic" << lightNumber << ": " << uniformIndex << endl;

}

*(The above ‘cout’s can be used to check if the values are being passed in and used in calculations, if the values return as ‘-1’ on run, then there is an error regarding that, check for typos or slightly different variable names and the like as they are the most likely causes.)*

*Because the function has been changed to include a new parameter in the .cpp file, the change will need to be reflected in the .h file too:*

void setLight(const GLuint program, const lightStruct light, GLuint lightNumber);

*After that, setLight can now be used to set up multiple lights, head back to main.cpp and look into the ‘init’ function. The original setLight call will need to be edited to contain the appropriate lightNumber value. While you’re at it, create new calls to set up two new lights:*

rt3d::setLight(shaderProgram, light0, 0);

rt3d::setLight(shaderProgram, light1, 1);

rt3d::setLight(shaderProgram, light2, 2);

*Now, you can set up new structs for the new lights, just like the first light is set up:*

rt3d::lightStruct light0 = {

{1.0f, 0.0f, 0.0f, 1.0f}, // ambient

{0.0f, 0.0f, 0.0f, 1.0f}, // diffuse

{0.0f, 0.0f, 0.0f, 1.0f}, // specular

{-5.0f, 2.0f, 2.0f, 1.0f}, // position

1.0f, //Attenuation - Constant

0.09f, //Attenuation - Linear

0.032f //Attenuation - Quadratic

};

rt3d::lightStruct light1 = {

{ 0.0f, 1.0f, 0.0f, 1.0f }, // ambient

{ 0.0f, 0.0f, 0.0f, 1.0f }, // diffuse

{ 0.0f, 0.0f, 0.0f, 1.0f }, // specular

{ 5.0f, 2.0f, 2.0f, 1.0f }, // position

1.0f, //Attenuation - Constant

0.09f, //Attenuation - Linear

0.032f //Attenuation – Quadratic

};

rt3d::lightStruct light2 = {

{ 0.0f, 0.0f, 1.0f, 1.0f }, // ambient

{ 0.0f, 0.0f, 0.0f, 1.0f }, // diffuse

{ 0.0f, 0.0f, 0.0f, 1.0f }, // specular

{ -5.0f, 2.0f, 12.0f, 1.0f }, // position

1.0f, //Attenuation - Constant

0.09f, //Attenuation - Linear

0.032f //Attenuation – Quadratic

};

*Lights 0, 1 and 2 will emit red, green and blue light respectively. In order for better results, change the values in the materialStructs just below the lights you have just edited so that they are white, and you can see the effect of each light more clearly:*

rt3d::materialStruct material0 = {

{1.0f, 1.0f, 1.0f, 1.0f}, // ambient

{0.0f, 0.0f, 0.0f, 1.0f}, // diffuse

{0.0f, 0.0f, 0.0f, 1.0f}, // specular

1.0f // shininess

};

*Nearly there*