*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

****

Graphics Programming M31622944 - 18/19

Coursework

Samuel peter stewart, s1832288

2019

Contents

[ADS Lighting 2](#_Toc8389656)

[Shader code 2](#_Toc8389657)

[Uniforms 3](#_Toc8389658)

[Basic Normal map 3](#_Toc8389659)

[Shader code 3](#_Toc8389660)

[Uniforms 4](#_Toc8389661)

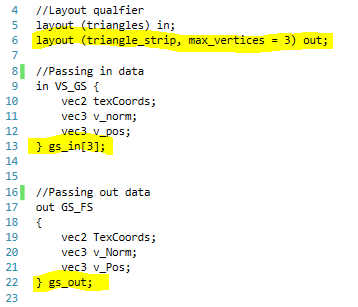
[Inverted colour Diffuse 4](#_Toc8389662)

[Shader code 4](#_Toc8389663)

[Uniforms 5](#_Toc8389664)

# ADS Lighting

## Shader code

Vertex shader (*ADS.vert*) – Texture coordinates, vertex normals, and vertex positions are all used in the fragment shader, hence, they are all brought in by layout qualifiers, and passed via the VS\_GS (vertex shader to geometry shader) interface block to the geometry shader. The vertex positions are multiplied by the ‘transform’ matrix which represents the MVP matrix, converting them to screen space. These translated positions are used to set gl\_Position and are also sent in the interface block.

Geometry shader (*passThrough.geom*) – This is a simple pass-through geometry shader that copies data from the VS\_GS block to the GS\_FS block. Because this application uses triangles, 3 vertices (and their associated normals and coordinates) are taken in at a time and passed to the fragment shader – to do this, a for loop (i < 3) is used, calling EmitVertex() at the end of each of the 3 loops, calling EndPrimitive() afterwards (the primitive being a triangle).

Fragment shader (*ADS.frag*) – ADS lighting involves applying ambient (A) colour/light, diffuse (D) shadows, and specular (S) highlights. First, the shader gets the initial fragment colour via the texture coordinates from the interface block (fs\_in.TexCoords) and stores it in the vector4 finalColour, which will then be modified with ADS lighting:

Figure 1 showing an array of 'in' blocks compared to a single 'out' block

Ambient lighting is applied first (it could be applied later if one wants to change the colour of the specular highlight or diffuse shadow). The sum of finalColour (the colour from the texture) and ambColour (the desired ambient colour) is halved to get the colour between these two values, and this becomes the new finalColour.

For diffuse lighting, the intensity of the effect on the fragment is calculated (as a float) as the dot product of the lightDir (the normalized direction from the surface to the light source) and the surface normal, updated to factor in the position/rotation of the model (calculated as the transpose of the inverse of the model matrix multiplied by the normal). If there is more than 90 degrees between the two values, the dot product will be 0 or less, meaning if we multiply the RGB values of finalColour by the diffuseIntensity, we will make them black (i.e. when a normal is facing away from the light, the fragment will appear dark) and if the surface doesn’t directly face the light, it will be made slightly darker. diffuseIntensity is capped at 1, so that the fragment can’t get brighter from diffuse lighting.

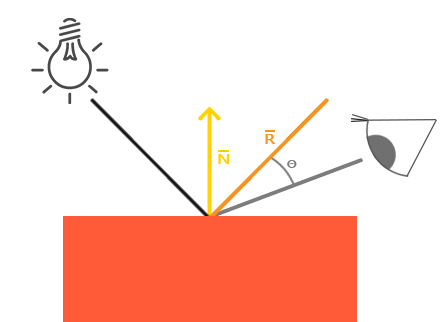
Specular lighting produces a bright highlight at points in the surface where the light would be reflected off the surface directly at the viewer. To simulate this effect, we get the angle of the viewer to the fragment and compare it to the perfect angle of reflection from the light source across the normal; the higher the dot product of these values, the greater the specular impact. The view angle is simply the normalized vector position, as this position is given relative to the camera (it is in view space) and the reflect() function is used to calculate the perfect reflection angle. When defining the float specularIntensity we use max(dot product, 0) to ensure a value less than 0 is not generated (specular lighting shouldn’t make anything darker) and this value is magnified by the power of 2.5 – this adjusts how sharp the highlight is, as it means a high value will be exponentially larger than a low value. We create a vector4 based on the specularIntensity value and add it to the finalColour (the specularIntensity value is first scaled down greatly so that it isn’t too intense). finalColour is then output as fragcolor.

Figure 2 showing the basis of specular lighting  
 image from learnopengl.com

## Uniforms

*Mat4 transform –* The MVP matrix used to convert vertex positions to view space, generated using the camera and the transform information.

*Mat4 u\_mm –* The model matrix based on the desired position, rotation, and scale of the mesh, used to update the normals.

*Vec3 lightDir –* The direction vector of the light. This is changed in MainGame.cpp with hard-coded values\* which is normalized in the shader.

*Vec3 ambColour* – The RGB value that is used for ambient lighting. This is changed in MainGame.cpp with hard-coded values\*.

*Sampler2D diffuse* ­– A link to the bound texture.

\* *lightDir and ambColour transition between two states based on the ‘daytime’ Boolean. There is a target value that each should be at by the end of their stage when daytime is true and another value when daytime is false. At the start of the stage, the s2Counter float will be 0, and by the end it will be 2, and we use this value to transition between the states. E.g.When daytime is true, the R value of ambColour should go from 0.4 to 0.98 to create the effect that the sun is setting, casting an orange/red light, so we take the starting value of 0.4 and add half of the difference between the two values multiplied by the s2Counter (the value is halved because the final s2Counter value is 2), so at the start, we are adding nothing (0.28 \* 0) and at the end we are adding the whole difference (0.28 \* 2) taking us to the target value.*

# Basic Normal map

## Shader code

Vertex shader (*shaderNormalMap.vert*) – Texture coordinates are used in the fragment shader, and vertex normals would be used in a more advanced version of the fragment shader, so both are brought in via layout qualifiers and passed out. Vertex positions are not used in the fragment shader, so they are not passed through.

Fragment shader (*shaderNormalMap.frag*) – Normal mapping involves using texture information to modify a normal (possible as both of these are made of 3 values; RGB and XYZ) This version of a normal map shader is rudimentary, and so does not utilize a BTN matrix to combine the normal map input with an updated surface normal (or the surface normal at all, for that fact) instead, it relies upon the rendered surface pointing directly at the camera (an effective normal of 0, 0, -1). Some of the code that would be needed to incorporate the surface normal has been included in the event that the shader is developed further: the normal is updated using the model matrix (as described in the diffuse lighting section above) and is converted from a value ranging from ‘-1 to 1’ to ‘0 to 1’ (this is done by multiplying by 0.5 and adding 0.5, which would result in 0 from -1, 1 from 1, and an appropriate value for anything in between) which is done because the normal mapped values cannot be negative, as they are RGB values (0.5 is made to represent 0, with anything below that being negative). The updated surface normal would now be combined with the value from the normal map, but for this shader we set the effective normal to be the exact value from the texture, before converting to a value between ‘-1 and 1’ (if we didn’t do this, all X and Y values would be positive which would mean they all ‘lean’ towards one direction when compared to the directional vector of the light which ranges from -1 to 1) and the ‘normal’ value is ready to be used.

An unmodified copy of the texture information is now used to check how the fragment should be rendered. If the texture is cyan, then we set the final colour’s alpha value to 0, making it transparent, otherwise, we make the final colour a neutral grey (in lieu of using a second texture for diffuse colour) and calculate the impact diffuse lighting -as described in the *ADS.frag* section above- using the dot product of the lightDir uniform and the newly generated normal. The fragcolor output is then set to the final colour (transparent, or diffuse grey).

## Uniforms

*Mat4 transform –* The MVP matrix used to convert vertex positions to view space, generated using the camera and the transform information.

*Mat4 u\_mm –* The model matrix based on the desired position, rotation, and scale of the mesh, used to update the normals.

*Vec3 lightDir –* The direction vector of the light. This is changed in MainGame.cpp by calculating the direction vector between the light source (which moves with a varying transform) and the ‘window’ object (with a fixed position of 0, 0, 30). Because the light source is drawn right before the window, we can send the uniform information before the transform is updated with the window’s position and still holds the light source’s position (meaning this value doesn’t need to be stored seperately).

*Sampler2D diffuse* ­– A link to the bound texture.

# Inverted colour Diffuse

## Shader code

Vertex shader (*shaderNormalMap.vert*) – *See above.*

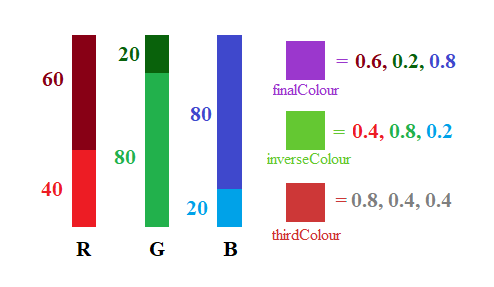
Fragment shader (*shaderColourDiffuse*) *–* The final colour is first set to the uniform value of inColour, and from this we get the inverted version of that colour (inverseColour) by subtracting the R, G, and B values from 1 (blue (0,0,1) -for example- would become yellow (1,1,0)). We then check the colour of the texture at this point, and if the texture is not white, we apply the coloured diffuse effect: like with regular diffuse, we update the normal, calculate the dot product between this normal and the normalized light direction, and limit this diffuseIntensity value between 0 and 1. Unlike normal diffuse, the finalColour is not multiplied by diffuseIntensity, making it brighter/darker, but rather we subtract the difference between this colour and the inverted version multiplied by diffuseIntensity from finalColour – this means when diffuseIntensity is 1 we subtract the whole difference between the two values, meaning the new value is equal to inverseColour, and when diffuseIntensity is 0 we are not subtracting anything, leaving finalColour as it was (any value in between will result in a colour in between the original finalColour and inverseColour, meaning the colours will transition smoothly across a curved surface\*). If the colour of the texture *is* white at that fragment, then the diffuse effect is not desired. Instead, we set the finalColour to a third colour that should be visually distinct from the other two – to achieve this, the finalColour is subtracted from the inverseColour, and this value is made absolute so that there are no negative values. In testing, this colour was often too similar to one of the other colours, so it is also inverted.

Figure 3 showing colour inversion and subtraction, and how they impact finalColour, inverseColour, and thirdColour

\* *This transition is present on the monkey mesh, but not the man mesh. It is possible this is due to the man mesh having less varied normals, but the exact reason is currently unknown.*

## Uniforms

*Mat4 transform –* The MVP matrix used to convert vertex positions to view space, generated using the camera and the transform information.

*Mat4 u\_mm –* The model matrix based on the desired position, rotation, and scale of the mesh, used to update the normals.

Vec3 lightDir - The direction vector of the light. This is changed in MainGame.cpp with hard-coded values which is normalized in the shader.

Vec3 inColour – The colour used set finalColour. This value is hard coded in MainGame.cpp as the worldColour value which is also used to colour the background/environment – because of this, the shader makes part of the model disappear completely into the background.