Graphics Programming

Software Development for Computer Games

Bill Mitchell

S1621554

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.

Bill Mitchell.

Contents

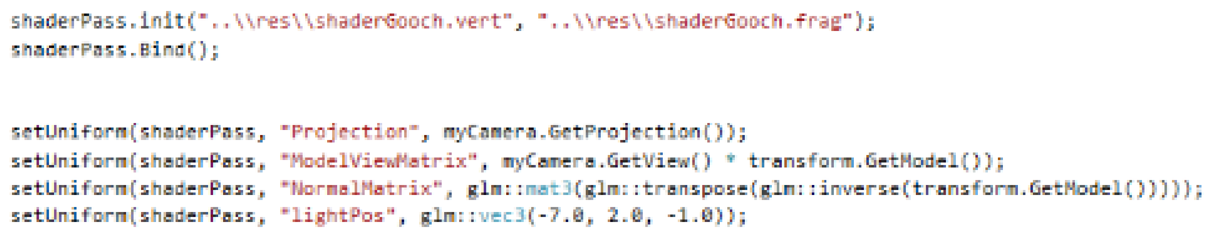
1. Introduction

Gooch shading is non-realistic rendering technique designed by Amy Gooch used to shade objects using cool and warm colours to affect the colour of the model depending on which side is affected by a light source, with warm being the colour affected by the light and cool being the colour away from the light. Gooch shading also makes use of Specular lighting to apply a highlight to the object.

This document will outline the code used to create the Gooch shader including; the DrawGame() function within MainGame.cpp, the shaders shaderGooch.vert and shaderGooch.frag.

2. Gooch Shader

2.1 Main Game



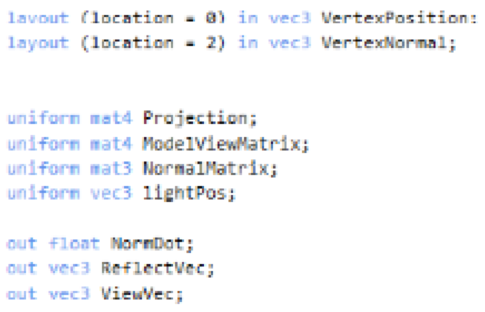
```
shaderPass.init("..\\res\\shaderGooch.vert", "..\\res\\shaderGooch.frag");
shaderPass.Bind();


setUniform(shaderPass, "Projection", myCamera.GetProjection());
setUniform(shaderPass, "ModelViewMatrix", myCamera.GetView() * transform.GetModel());
setUniform(shaderPass, "NormalMatrix", glm::mat3(glm::transpose(glm::inverse(transform.GetModel()))));
setUniform(shaderPass, "lightPos", glm::vec3(-7.0, 2.0, -1.0));
```

Figure 1. drawGame() within mainGame()

The shader is initiated in the in the DrawGame() function of MainGame.cpp by calling the init() function of the shader script and passing in the location in the file of the shaderGooch fragment and vertex files. The shader is then bound, and the Uniforms are then passed in by making use of the SetUniform function, which uses overloads to pass in multiple types of variables, within MainGame.cpp. Several variables are passed into the shader including the shader, along with the name of the Uniform within the shader and the value of which that uniform should be set.

2.2 shaderGooch.vert

2.2.1 Variables

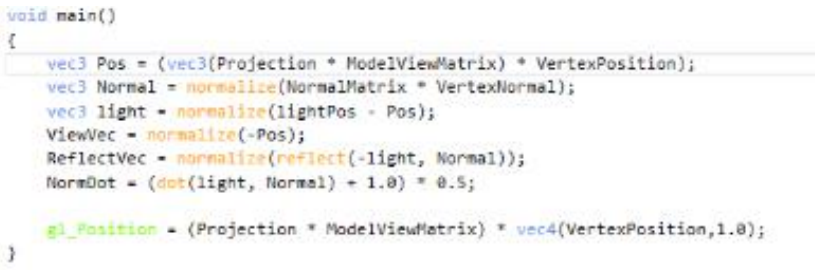


```
layout (location = 0) in vec3 VertexPosition;
layout (location = 2) in vec3 VertexNormal;


uniform mat4 Projection;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform vec3 lightPos;

out float NormDot;
out vec3 ReflectVec;
out vec3 ViewVec;
```

Figure 2. Variables within the vertex shader

The variables within shader.vert are; two Vector3's, VertexPosition and VertexNormal, which are bound to a location when the shader is initialised and called from the location. Two Matrix4's, Projection (which holds the projection matrix) and ModelViewMatrix, a Matrix3, NormalMatrix, and a Vector3 lightPos (which holds the position of the light) are passed into the shader as a uniform from MainGame as discussed earlier. There are also three variables set as out, which allow the variables to be passed to the fragment shader, or other files within the shader. These variables are NormDot, a float, ReflectVec and ViewVec, both Vector3's.

2.2.2 main()



```
void main()
{
    vec3 Pos = (vec3(Projection * ModelViewMatrix) * VertexPosition);
    vec3 Normal = normalize(NormalMatrix * VertexNormal);
    vec3 light = normalize(lightPos - Pos);
    ViewVec = normalize(-Pos);
    ReflectVec = normalize(reflect(-light, Normal));
    NormDot = (dot(light, Normal) + 1.0) * 0.5;

    gl_Position = (Projection * ModelViewMatrix) * vec4(VertexPosition,1.0);
}
```

Figure 3. main() within the vertex shader

The main function within the vertex shader is used to set the values that are to be passed out to the fragment shader. This is done by first creating a local vector3 called Pos that is equal to a vector3 of the ModelViewProjectionMatrix multiplied by the VertexPosition which gets the position of the vertex.

Then a Vector3 called Normal is set to the NormalMatrix multiplied by the VertexNormal and normalized to give the normalised version of the Normal vector.

The final Vector3 to be created is light and it is the lightPos minus the Position of the vertex and normalized to get the position of the light in comparison to the particular Vertex of the model.

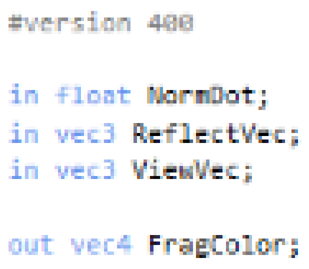
The View Vector is given by the negative of the Pos vector, normalized. The Reflect Vector is set using the reflect function passing in the negative of the light vector and the Normal variables and then this is normalized.

Finally, the NormDot variable is set which is the dot product of the Normal and light variables which gives the weighting value of the light. These three variables are passed to the fragment shader to be used there.

gl_Position is then set to the ModelViewProjection Matrix and multiplied by the VertexPosition which sets the position of the vertex as a Vector4 rather than as a Vector 3 set in the Pos variable.

2.3 shaderGooch.frag

2.3.1 Variables



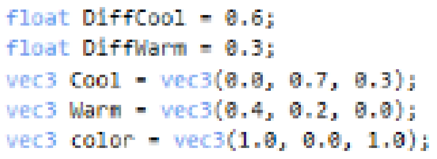
```
#version 400

in float NormDot;
in vec3 ReflectVec;
in vec3 ViewVec;

out vec4 FragColor;
```

Figure 4. Variables within the fragment shader

Within the fragment shader, there are three variables that are passed in from the vertex shader (and are named the same to allow for the values to be passed to the right variable). These variables hold the Dot product of the Normal and the Light vectors of vertex shader (NormDot) which is the weighting of the light that will affect the vertex, the reflection vector and the view vector.

The final variable, FragColor, is used to determine and return the colour that should be applied to the vertex fragment.

2.3.2 main()

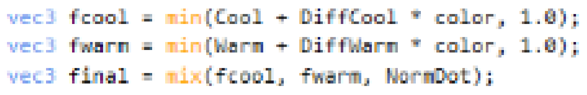


```
float DiffCool = 0.6;
float DiffWarm = 0.3;
vec3 Cool = vec3(0.0, 0.7, 0.3);
vec3 Warm = vec3(0.4, 0.2, 0.0);
vec3 color = vec3(1.0, 0.0, 1.0);
```

Figure 5. Local Variables within main() of fragment shader

Within the main function of the fragment shader, several variables are set up to be used to get the final colour; DiffCool and DiffWarm are Floats that determine how the fragment colour will be affected by the Cool and Warm vectors in order to apply the two colours to the object.

Cool, Warm and Color are all Vector3's that hold the colours to be used for the object with Cool being the colour that is away from the light, Warm is the side of the object hit by the light and color is the base colour that will be used to mix with the Cool and Warm colours.

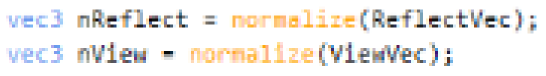


```
vec3 fcool = min(Cool + DiffCool * color, 1.0);
vec3 fwarm = min(Warm + DiffWarm * color, 1.0);
vec3 final = mix(fcool, fwarm, NormDot);
```

Figure 6. Local Variables using OpenGL functions within main() of fragment shader

The Vector3's fcool and fwarm are used to get the different colours that the fragment will be if it is hit by the light and if it is not. It makes use of the min() function, which returns the lowest of two values, to set the value to either the colour value of the Cool vector added with the color multiplied by the DiffCool variable to get how much the base colour is affected by the Cool vector (or the Warm vector and DiffWarm for fwarm) or to the shade white if the value of the sum is more than 1.

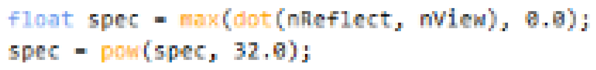
The Vector3 final makes use of the mix() function to interpolate between the fcool and fwarm vectors (the fragment shades if affected by the warm or cool vectors) and the NormDot (the weighting variable of the light) variable, which is the values used to interpolate between. This is used to give the final colour for the fragment as affected by the light.



```
vec3 nReflect = normalize(ReflectVec);
vec3 nView = normalize(ViewVec);
```

Figure 7. Local Variables within main() of fragment shader using normalize()

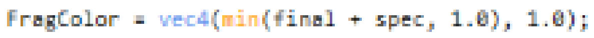
A Vector3 called nReflect is created and used to store the normalised version of the passed in ReflectVec which is set by using the normalize() function and the Vector3 nView holds the normalised version of the ViewVec passed in.



```
float spec = max(dot(nReflect, nView), 0.0);
spec = pow(spec, 32.0);
```

Figure 8. spec being set using OpenGL functions

A float called spec takes the nReflect and nView vectors and uses them to get the Dot product, using the dot() function, and takes the maximum values between the dot result and 0. This is done using the max() function. The spec variable then is set to itself to the power of 32, making use of the pow() function. This determines how much specular lighting will affect the fragment and be added to the final color.



```
FragColor = vec4(min(final + spec, 1.0), 1.0);
```

Figure 9. FragColor

Finally, the FragColor is set by taking the minimum value, using the min() function, between the final (the colour of the fragment) and spec (the specular lighting affecting the fragment) variables added together or set to 1 to give the fragment a white colour and setting this to a Vector4 to give an RGBA variable for the vertex, with the 1.0 outside the min() function being the alpha and the minimum result being the RGB.

3. References

Benton, A (2012) Advanced Graphics. Available at: http://bentonian.com/teaching/AdvGraph1314/7.%20OpenGL%20and%20shaders%202.pdf [Accessed on: 02/05/2019].

3.1 Models

Cylinder - https://free3d.com/3d-model/hollow-cylinder-v1--917904.html

Pokeball - https://free3d.com/3d-model/pokeball-4613.html

Hexagon – https://free3d.com/3d-model/hexagonal-prism-v1--290091.html