Lab 09 Report - Gipson

# Introduction

Shaders are the foundation of game graphics. Small programs ran on the GPU to tell the display what it needs to, well, display. Using shaders you can create all kinds of strange and awesome effects. Aswell as simply putting a color on the screen. Lighting, textures and colors are three of the basic building blocks for shaders, and in this lab I will learn how to use all three.

# Methods

For shaders you always need a few things. Actually some of these may be optional, but its best to make use of them. These methods are for unity shaders in specific, using CG as much as possible. Start out by declaring a shader block. The Shader keyword is followed by a string, this string is the name of the shader. By using a “/’’ you can nest a shader and organize. Next you will want to declare a properties block, this holds all of the information used to interface with unity. This is a shader lab thing. Properties follow the following syntax:

Name(“display name”, type) = defaultValue

Common naming convention is \_PascalCase. You can have colors, textures, ranges, ints, bools, floats and a few other types of properties. Look at the documentation for a full list. After the properties block you need to declare a subshader block. This is where your actual shader code goes. You can have multiple subshaders, and the gpu will select which is best for the device running the shader. This is good for multiplatform games. Inside the subshader you can choose to have a tags block. This is to help bring in information from unity. Tags syntax is:

“name”=”value”

There are preset names and values for shaderLab, see documentation for further details. If you want to do lambert lighting, use Tags{"LightMode" = "ForwardBase" } . now we move to the pass block, shaders can have multiple passes. These run in order and are used to build up effects. Say a base color followed by a lambert lighting pass. Within the pass is where all of your calculations will go.

Start your pass with CGPROGRAM and end it with ENDCG – these are special tags telling the compiler what language is being used so that it can process it properly. CG requires pragmas for its functions that need to be ran automatically.

#pragma vertex vert

#pragma fragment frag

Are how I declare my vertex and fragment functions. Pragma syntax is:

#pragma functionType functionName

With the pragmas declare you need to declare any of your user variables, aswell as give the pass a link to the properties you declared earlier. Redeclare the variable as uniform type name. the name must be the same exact name as the property for unity to make the connection. Color is a unity variable, but not a cg variable type, so if you have a color declare it as float4.

Next make your input and output structs. Input is what will be filled by unity. Use symantics to accomplish this. There are a lot of symantics available, POSITION is the most important in our case. Look to documentation for others. In the output synmatics are used to put the data back to the proer places. SV\_POSITION will be used to return the position, but in a way that directX can handle it and send it to the proper places across different platforms.

Now you will need to write your vertex and fragment functions. Use the names you declared in the pragmas. To get the position needed inside of your vertex function use the mul function with UNITY\_MATRIX\_MVP and your input.position.

In your fragment function just return color. This will allow you to have flat color shading. If you return texture you can have a texture on your object.

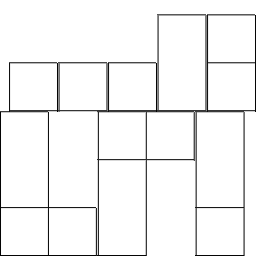
By adding different calculations to your vert and frag functions you can make your shader do all the fun things shaders are known for. These functions are where it all takes place.

# Conclusion

Shaders are really cool, but also finicky and the syntax isn’t the greatest. There is a lot that goes into shaders but once you understand the building blocks you can make most anything you wish. I enjoy shaders a lot and can’t wait to learn more.

# Post-Lab

1. Up map this object.



* 1. Lots of seams, and horribly packed. But I’m a programmer, what do I know…

1. What meaning does a UV have in relation to a texture and/or a model?
   1. It is the 2d coordinates of a 3d mesh that the texture uses for proper placement
2. What does it mean when you say that you will sample a texture?
   1. Get the color information
3. What does it mean when you say that you are unwrapping a texture?
   1. Mapping it to a 3d mesh, but you do not unwrap a textures; you unwrap meshes to map them to 2d space, a.k.a. UV space.
4. If I had a sampler2D named \_RawrTex, what would I name the next variable to capture the tiling and offset of that sampler2D?
   1. \_RawrTex\_ST

# Code

**Shader** "Lab09/texture" {

**Properties** {

\_Color("Color Tint", Color) = (1,1,1,1)

\_MainTex("Diffuse texture", 2D) = "white"{}

\_Attenuation("fall off", Range(0,5)) = 0

}

**SubShader** {

**Pass**{

Tags{"LightMode" = "ForwardBase" }

**CGPROGRAM**

#pragma vertex vert

#pragma fragment frag

//user -color

uniform float4 \_Color;

//user -texture

uniform sampler2D \_MainTex;

uniform float4 \_MainTex\_ST;

//user -lambert

uniform float \_Attenuation;

//unity variables

uniform float3 \_LightColor0;

struct input{

float4 vertexPos : POSITION;

float4 textureCoord : TEXCOORD0;

float3 vertexNormal : NORMAL;

};

struct v2f{

float4 pixelPos : SV\_POSITION;

float4 tex : TEXCOORD0;

float4 color : COLOR;

};

v2f vert(input i){

v2f toReturn;

//texture

//get pixel position

toReturn.pixelPos = **mul**(UNITY\_MATRIX\_MVP, i.vertexPos);

//get texture coords

toReturn.tex = i.textureCoord;

//lighting

float3 lightDirection;

//get ambient light color

float3 ambientLight = UNITY\_LIGHTMODEL\_AMBIENT.rgb;

//get ight direction

lightDirection = **normalize**(\_WorldSpaceLightPos0.xyz);

//get normal

float3 tempNorm = i.vertexNormal;

//convert to objSpace

float4 objNorm = **mul**(float4(tempNorm, 1.0), \_World2Object);

//normalize

float3 normalizedNormal = **normalize**(objNorm).xyz;

//dot product of light color normals and light direction aswell as falloff

float3 diffuseReflection = \_Attenuation \* \_LightColor0.xyz \* **max**(0.0, **dot**(normalizedNormal, lightDirection));

//ambient light added

float3 final = diffuseReflection + ambientLight;

//float4(xyz, w) float4(x,y,z,w)

toReturn.color = float4(final,1.0);

//toReturn.color = objNorm;

return toReturn;

}

float4 frag(v2f i) : COLOR {

float4 tex = **tex2D**(\_MainTex, \_MainTex\_ST.xy \* i.tex.xy + \_MainTex\_ST.zw);

return i.color \* \_Color \* tex;

}

**ENDCG**

}

}

//FallBack "Diffuse"

}