Shader’s and their application in games

Following American psychological association guidelines

In partial fulfillment of requirements of GPG230

Noaman Khalil

SAE Institute Dubai

Shader’s and their application in games

Abstract

The goal of this paper is to demonstrate the understanding shaders and their application in game development. The paper also aims to demonstrate the ability to deform meshes using an example shader with further development of the shader to emulate water waves.

Introduction

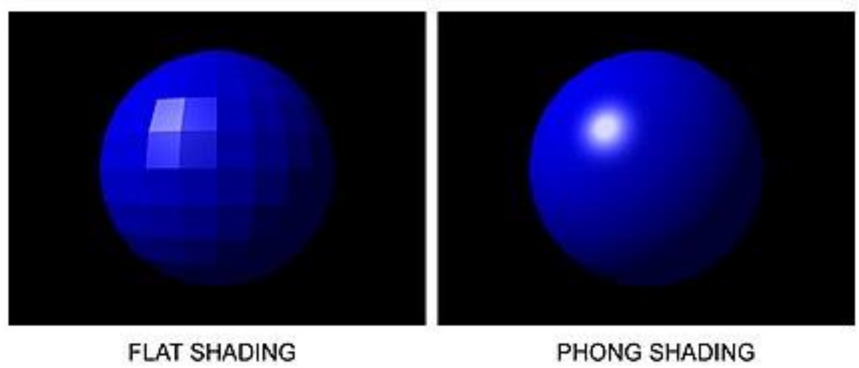
The simplest way to define it would be “A program that is executed on your graphics card” as per webopedia (("What is shader? Webopedia Definition," n.d.)), although albeit vague it is the simplest definition however this does not explain what a shader is and what its purpose is in regards to computer games. A shader by a much broader definition instructs the GPU on how objects are going to be rendered via shader code, this also s used to produce lighting and shader in 3D modelling.

Common uses

Shader’s are most commonly used in 3d models and games to emulate real life effects such as for example, water waves, these cannot be processed by normal code written in C# or JavaScript as they would execute on the central processing unit rather than the graphical processing unit which is much faster than the CPU and is more optimized for the GPU thus there are specialized languages made for writing shaders such OpenGL Shading language which is also known as the GLSL, another example of a specialized language would be DirectX high level shader language which is also called HLSL for short there are numerous such languages for more specific use cases like metal shading for Apple. ("Shading language," 2017)

Shader's and their application in games

The most common use for a shader is to produce lighting and shadow effects for 3d models, an example below is that of the pong shader which one of the earliest examples of shaders.



(Phong-shading-sample, n.d.)

The problem

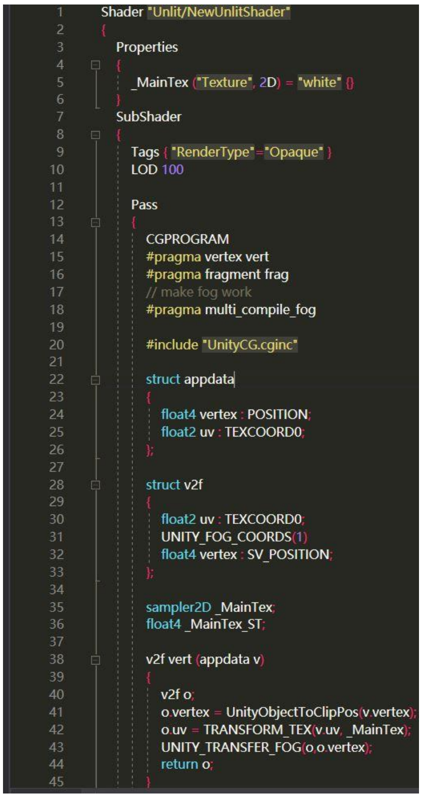
Shaders address a very important problems within games which are attributed to how games are usually made, for example shaders do calculations on the GPU which has significantly more processing power than that of the CPU which is good for doing smaller calculations but in a scenario where an object has for example a thousands of vertices and they are changed every frame then the amount of calculations done would number in the thousands which would cause a the framerate to be in the single digits which is very bad for games which have a target framerate of running on 60 frame per second or even 120 in the case of less demanding games and by doing these calculations on the GPU this can be achieved .

Shader code

The shaders aim is to demonstrate understanding of shaders and their application and thus I chose an unlit shader for the example as this would be the simplest base to work off for the demonstration, but before we do I will be explaining base first before explaining how the deformation takes place. The reason I chose this form of shader over many others is that the concept of mesh deformation is interesting and is something that I as a developer have done in the past in C#, thus the interest in doing it through shaders.

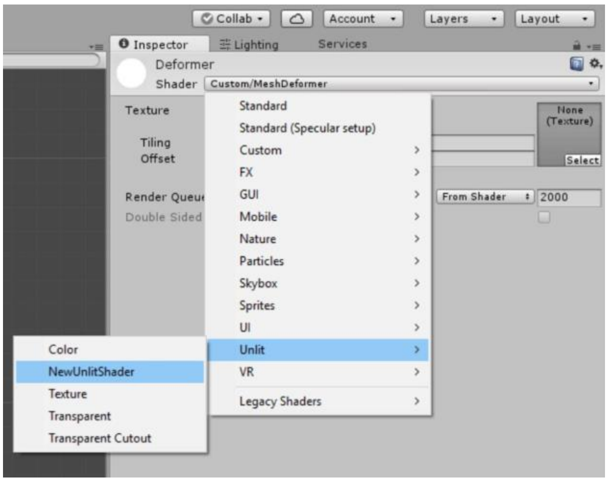
The base of this shader will be an unlit shader as it has all the minimum requirements needed to make our mesh deformation shader to emulate water waves.

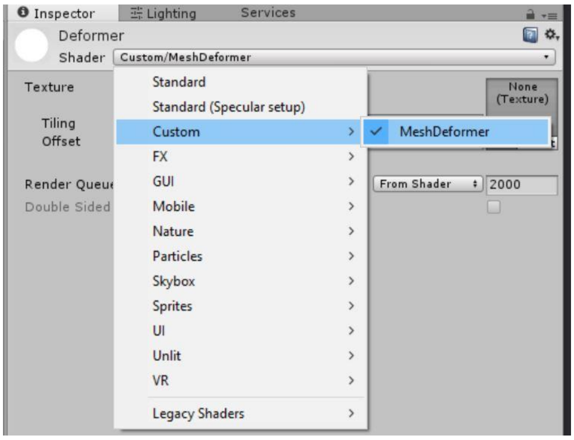
To start off we must create a NewUnlitShader which will act as a base for our mesh deformation shader, it should look like the images below in the editor and in the inspector windows in unity.



Next we must understand what each line does and what it means, the first line indicates the shaders name and which category it will be under when selecting it from the material windows which is also shown in the image below. For this example it is under the unlit

Shader subcategory however this will be changed to the custom category which is also shown below.

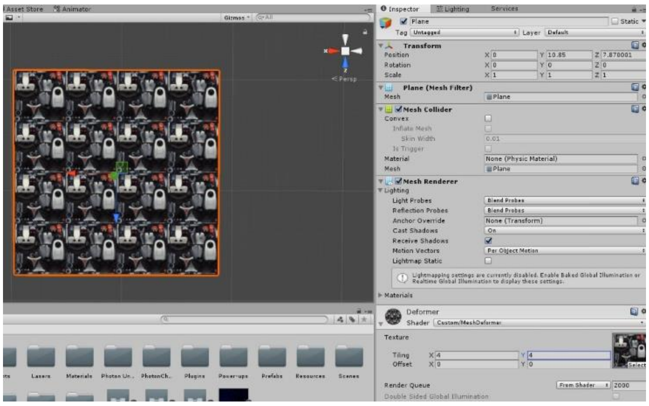




```
1  Shader "Custom/MeshDeformer"
2  {
3      Properties
4      {
5          _MainTex ("Texture", 2D) = "white" {}
6      }
7      SubShader
8      {
9          Tags { "RenderType"="Opaque" }
10         LOD 100
```

Next we must understand what the properties section of the code does, this is simply to handle public variables such as the main texture which can be seen on line 5, this main texture will be by default a white color however it will take the color which is defaulted by the texture .

Next we must test if this works and in order to do this I attached a random texture from the assets folder which looks like the image below. Another thing to note is that the image can be tilled in order to make a repeating pattern using the tiling option. The image on the right is tiled.



Next we must understand what a sub-shader is, each shader in unity consists of a list of sub-shaders. When the unity has to display a mesh it will find the shader to use, and will pick the first sub-shader that runs on the users graphics card. A shader can have multiple sub shaders for different platforms IE: PlayStation or IOS.

Next we must understand the use of tags in the sub-shader, which is used by the sub-shader to determine how and when they are to be rendered in the rendering engine, the RenderType tag arranges shaders into a few predefined gatherings, e.g. is a opaque shader, or an alpha-tested shader and so on. This is utilized by Shader Replacement and at times used to create produce camera’s depth texture.

```
SubShader
{
    Tags { "RenderType"="Opaque" }
    LOD 100
```

The LOD mentioned above represents the level of detail allowed by the shader however it is to be noted that that the LOD value only works with a value less than the given number. By default the level allowed for the LOD value in infinite as long as all the shaders are supported by the user’s hardware.

```
12      Pass
13      {
14          CGPROGRAM
15          #pragma vertex vert
16          #pragma fragment frag
17          // make fog work
18          #pragma multi_compile_fog
19
20          #include "UnityCG.cginc"
21
22          struct appdata
23          {
24              float4 vertex : POSITION;
25              float2 uv : TEXCOORD0;
26          };
27
28          struct v2f
29          {
30              float2 uv : TEXCOORD0;
31              UNITY_FOG_COORDS(1)
32              float4 vertex : SV_POSITION;
33          };
34
35          sampler2D _MainTex;
36          float4 _MainTex_ST;
37
38          v2f vert (appdata v)
39          {
40              v2f o;
41              o.vertex = UnityObjectToClipPos(v.vertex);
42              o.uv = TRANSFORM_TEX(v.uv, _MainTex);
43              UNITY_TRANSFER_FOG(o,o.vertex);
44              return o;
45          }
46
47          fixed4 frag (v2f i) : SV_Target
48          {
49              // sample the texture
50              fixed4 col = tex2D(_MainTex, i.uv);
51              // apply fog
52              UNITY_APPLY_FOG(i.fogCoord, col);
53              return col;
54          }
55          ENDCG
56      }
```

Next we must understand what the Pass block does, which is a single instructions to the GPU telling it what to do.

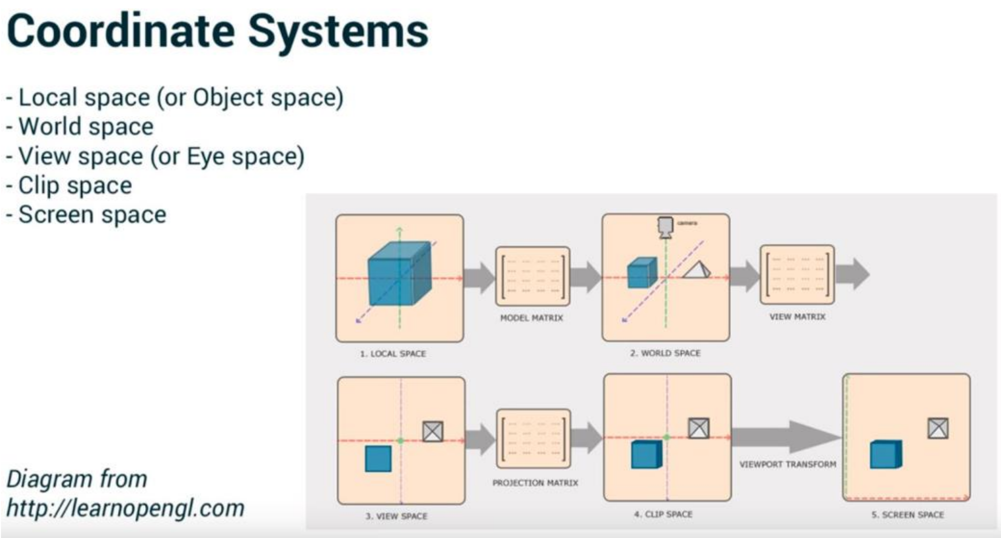
#Pragma are pre processer directives

The two struts are known as data objects which are later passed into the v2f vert & fixed4 frag on lines 38- 45 & 47- 54 respectively .

The appdata struct passes in information from the vertices in the form of a packed array which contains 4 floating point numbers and uses Symantec binding to tell the shader it is a position , similarly in the v2f strut in line 32 it does the same however this is a world positon . It is to be noted that the appdata contains its local coordinates in line 24. The float2 uv on line 24 is the texture coordinates in a 2d space.

The v2f vert uses its struct to make it easier to pass in information which is later used by the method, on line 40 a declaration is made to be used to convert the local position of the vertices into a world positon in order for it to be used by unity. This process requires it to be changed from local space to world space to view space then clip space and finally to screen space which can be visualized in unity.

The function used to do this is the “Unity object to clip position” used in line 41 which can only be accessed through the #Include syntax on line 20 which gives access to unity helper functions similarly to what MonoBehvaiour does for C# as there is no way to do inheritance in shader programming . An image below has been attached to illustrate the conversion process.



(Unity, 2017)

On line 42 the tiling takes place by taking in UV and the texture, this is what causes the tiling to take place.

The fixed4 frag does the same with its struct however uses a SV_target which is used for the frame buffer , line 50 samples color using a 4 floats for RGBA and uses a unity helper function “Tex2d ” to get the color and isntrcut the shader on how to draw each color pixel according the information provided by the UV struct .

Now that we understand the unlit shader, we must apply the instructions to deform the mesh, in order to do this we must declare a vector 3, this is going to be used to multiply the world positon of each vertices , next we must actually move each vertices to look like waves in order to do that we must access the vertices Y postion which can be seen in line 43 and add its position (worldPos.z) by time which is _Time.w in a sin pattern to emulate water waves or in other words , mesh deformation.

Shader's and their application in games

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    float3 worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
    o.vertex.y += sin(worldPos.z + _Time.w);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    UNITY_TRANSFER_FOG(o,o.vertex);
    return o;
}
```

Conclusion

Shaders are different, however they are fairly rewarding with the outcome they provide provided the developer can get over it not being like traditional programming with regards to inheritance the ability to autocomplete keywords or even have intelli-sense .

References

hong-shading-sample. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Shader

Shading language. (2017, October 12). Retrieved April 4, 2018, from https://en.wikipedia.org/wiki/Shading_language

Unity. (2017, June 27). *Writing Your First Shader In Unity - Anatomy Of An Unlit Shader [3/8] Live 2017/6/21* [Video file]. Retrieved from https://www.youtube.com/watch?time_continue=146&v=Tr9PLpj7Kzc

What is shader? Webopedia Definition. (n.d.). Retrieved April 4, 2018, from https://www.webopedia.com/TERM/S/shader.html