Shader Graph Implementations

Preface:

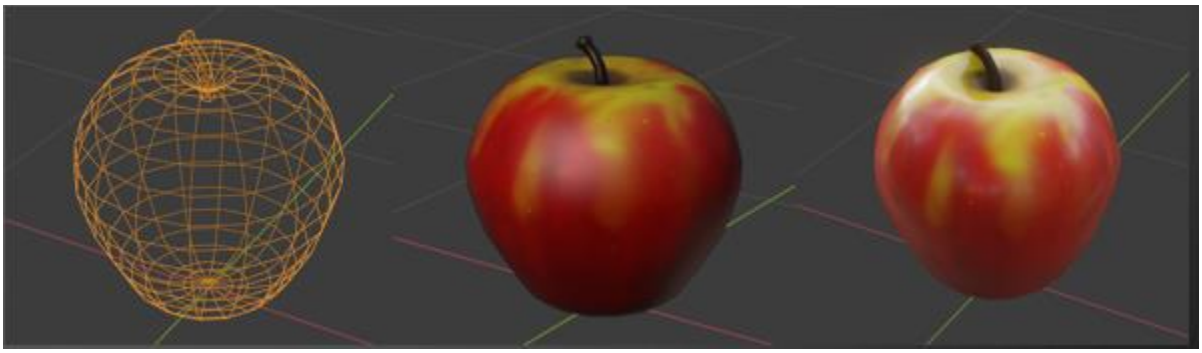
This document serves as an introduction to Shader Graph and the implementations of it that were used inside of the Unity environment for the Construction Safety Simulator Project. It is not a complete guide to what it is and how to use it.

3D Models:

3D models consist of two major parts. The first part is the mesh which is a collection of 3D points and their interconnections with each other. These interconnections form polygons, and the polygons can form more complex shapes. These collections of points and relationships mean absolutely nothing until we can make some visual representation on a computer screen with them (at least so far as computer graphics are concerned). This is the job of the second part of the 3D models. Typically, in modern 3D graphics, this can be resolved to a material, which holds information about what should be drawn on the computer screen. So, the mesh is a representation of where to draw, while a material is a representation of what to draw.

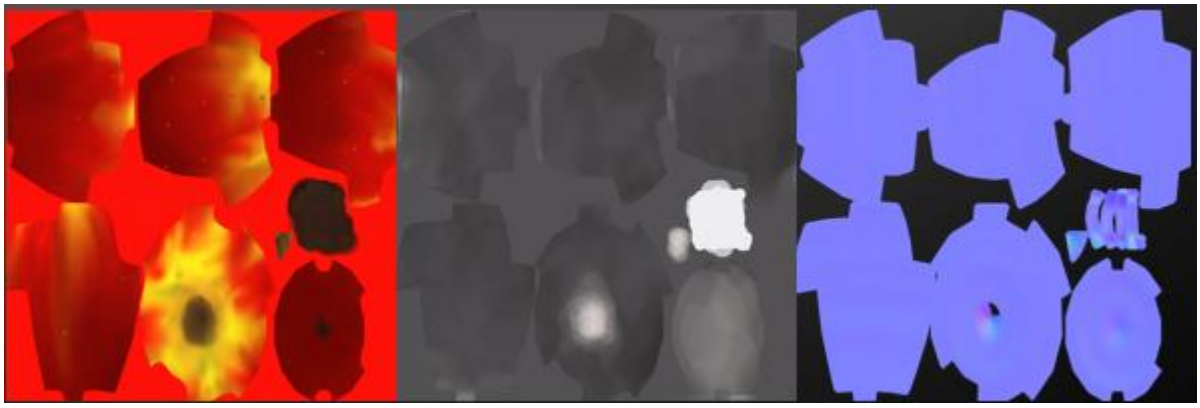
Realism of a model can be achieved through increasing complexity of either the mesh or the material assets. This is dependent upon the granularity that the viewer will be experiencing the model in. Generally, additions are placed on the mesh to allow for better viewing of the model in a zoomed-out format. When features of the model need to be expressed on a minute level, these are done with the materials for the model. Increasingly so in recent advancements, materials and physics-based rendering have trumped the need to increase the complexity of the mesh models. This is because techniques for physics-based light interactions on the surfaces of the model, along with apparent 3D morphing of the model can be achieved on the surface of a model using functional data operations instead of prescribed changes in model geometry. The advent massively programmable and pipelined GPUs has accelerated this.

Below is an example of a 3D object. You can see the transfer from a simple mesh outline to a simple material applied to the mesh, to a physics-based material which incorporates data about how the surface interacts with light in a 3D sense. The vast improvement in the realism of the object has nothing to do with the mesh: it is all conscribed within the functional rendering of colors on its surface that replicate real life events.



This increase in fidelity comes through applying successive layers of physical feature information to the material of the 3D asset. Below gives an example of this data for the above object. The first is a

texture map which is purely a color shading mechanism for the polygons on the mesh. The second is specular map which defines how much light bounces off the surface of the model at a given point on the model. The last map is a surface normal map which allows for changing the radial direction that the light reflects at off a polygonal surface. This allows for simulating microscopic surface deflections along an otherwise flat surface. The first map is used to render the middle image, while the second and third maps are used to render the final image.



The clear point to take from all of this is that the visual quality of an object has a massive dependency on the materials used to define the drawing of the model on the computer screen. This brings us to our next topic: shaders.

Shaders:

Like processes that run on a CPU, shaders are processes that run on a GPU. They are the algorithms that run to ultimately decide what appears on the screen for a computer graphics implementation. They take in data about the construction and orientation of a 3D model convert this to mappings of pixels on the screen. They also take in information about what colors the pixel at that point on the screen should be. This can be referenced back to the materials for the 3D object. The materials represent a logical grouping of the data, while the shaders are the actual processing of that data.

To achieve a higher level of realism requires that we be able to implement shaders on the GPU. Most of the time this is performed in a pre-built shader in Unity. However, there are occasions in which the shaders provided by Unity will not meet all our requirements, and so it becomes necessary to be able to make our own shaders. This process is cumbersome and necessitates having an in depth understanding of shader construction. Programming shaders is akin to programming in assembly, and the compilation process takes a long time to perform. This is where we can take advantage of the Shader Graph features of Unity.

URP and Shader Graph:

The Universal Render Pipeline (URP) is the newest pipeline for rendering computer graphics in Unity. It will be the standard for rendering in the Unity software moving forward, and there will be no more updated to the original Standard Render Pipeline (SRP) in Unity anymore. For these reasons alone it is more opportunistic to switch the project to URP. While I realize some developers have issues with URP, this is usually because they are using systems that only work with the older SRP system and are unwilling to move to the newer technology. The newer URP has been shown to run more efficiently than

the older system, provided that the technologies in the project utilize it in the correct fashion. Thus, the project was moved to the URP pipeline, and the assets were updated to use its methodologies. This involved updating the materials for the assets in the project to use the newer URP shaders. New models brought into the project will have their shaders mapped to Unity's URP shaders at this point. There may be some assets still in the project that have not been switched yet. This can be easily rectified by just switching them to a different shader in most cases. Each of the materials in Unity has a selector drop down at the top of the material in the editor view to accomplish this task. But in some instances, this cannot be done on the fly as the material is specialized. This typically occurs with asset prefabs that were downloaded off the Unity store and are not compatible with URP or Shader Graph. It may also be the case that we wish to create our own customized shaders for different visual effects. These things can be easily accomplished through Shader Graph.

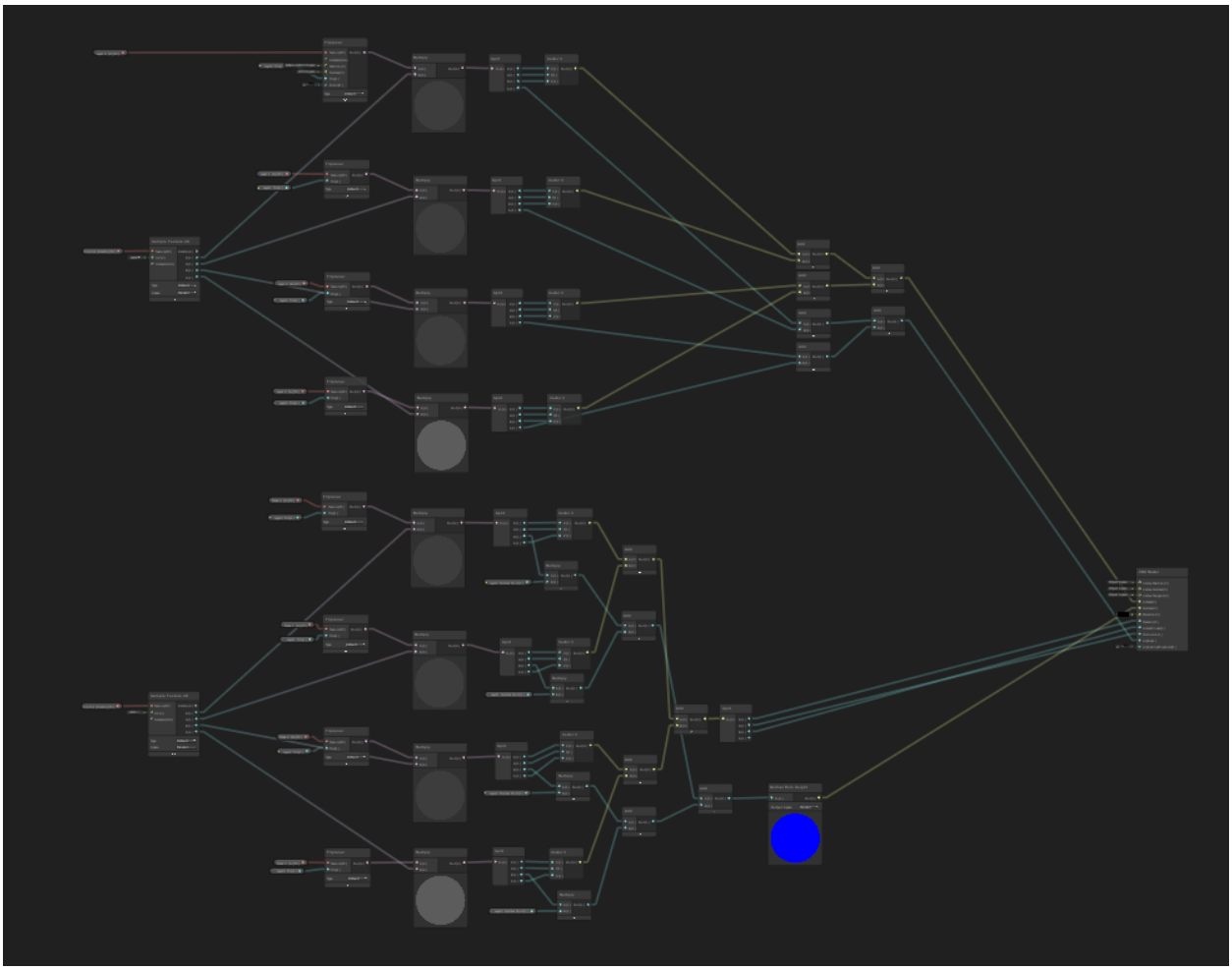
Again, shaders take input, operate on the input, and produce some output. When we are creating a Shader Graph shader that is designed as a replacement for an older shader, we simply need to look at the older shader to get an idea of what it did, then replicate this in the new Shader Graph shader. We first open the old shader and find all the input vectors for that shader. In the new Shader Graph shader, we define these same inputs with the same type and name as were in the original shader. Now the original models that used the old shader can interface with the new shader with no issues as they represent the same interface. We then look at how the inputs of the old shader were used inside of its program. We then replicate this by applying successive nodes in the new Shader Graph shader to get the respective processing that was available inside of the old shader. This completes the implementation of the new shader. The new shader does not require any compilation and can be changed on the fly and debugged within the Unity Editor environment. This is a much more efficient way to make shaders in the long run. We will next describe the specific shaders we created to solve some common issues we found while building the Construction Safety Simulator.

Terrains and Tri-Planar Shader:

Terrains were implemented in the Construction Safety Simulator instead of predefined meshes for several reasons. Terrains have simple editing tools designed specifically for building terrains very rapidly on the fly within Unity. Comparatively, predefined meshes take much longer to build. Additionally, terrains are much more efficient to render that predefined meshes as the Unity system will automatically reduce model complexity with far off objects.

An abnormality was noticed with high resolution textures on the terrains surface. The project requires placing ditches on the terrain. On the verticals to the ditches, it was found that the shader used for the terrain objects was stretching the texture in a way that was not conducive to a realistic environment. Therefore, we built a new shader using Shader Graph that had a feature known as tri-planar mapping. This stops the stretching of textures over the verticals of the terrain. Here is a side-by-side example. The new shader is on the left, the old on the right:

You can see how the new shader does not stretch the textures used on the terrain. To perform this, we used the same process detailed above. We looked at the old shaders inputs and created exactly those input on the new shader. We implemented the functions of the old shader but applied the tri-planar mapping method within the new shader so the textures would not be stretched. Here is the final shader:

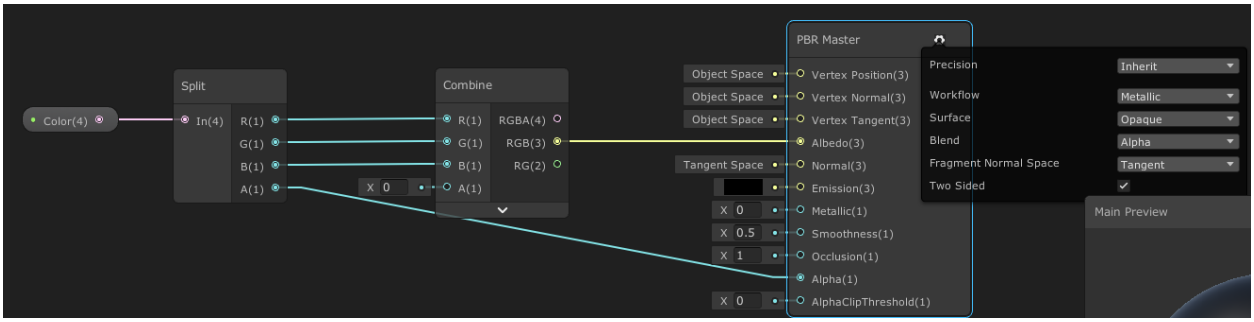


This can be opened in Unity's Shader Graph for analysis. Essentially, the normal method of mapping a texture onto a 3D surface has been swapped out for using a tri-planar method, in which the scaling factors for each of the dimensions are 1. This makes the textures wrap around the objects regardless of the usage of a height map (which is what the Terrain objects use to create height differences). It may look large, but this is only because the mapping technique needed to be applied to all the information that could be input to the standard physics shader in URP. This means textures maps, normal maps, specular maps, metallic maps, and any number of other maps. You will notice a lot of duplication in the graph structure. This is because the same process is applied to each of these types of maps. Another example of a shader that was produced with Shader Graph was used on extremely simple models that lacked a closed surface in geometry.

No Backface-Culling Shader:

Certain objects that were imported into Unity were highly simplified. This meant that the surface of a polygon on the model needed to be viewable on both sides. This is an issue as most 3D graphics programs employ a technique known as back face culling in order to reduce the number of calculations that need to be performed in each frame. If a polygon has its surface normal pointed away from the camera view, then it is said that its back face is facing the camera. In this case a back face culling pass will simply remove this polygon from further processing. For our simplified 3D objects, this

makes the geometry appear invisible when viewed from certain angles. In order to correct this, a new shader was developed that essentially turned off the back face culling feature of the standard URP physics-based shader. Here is the graph of the shader (the shader is called NoBackFaceCulling):



This is essentially a standard shader, however if you look at the final rendering node you can see that 'Two Sided' has been selected, which means that the renderer will no longer perform back face culling, and the polygons on simple models can be seen from both sides. This is slightly less efficient, but the models it applies to were so simple that the performance hit is unrecognizable.