Graphics Assignment Implementation Report

* I have used the skinning lab project as a base project for this assignment, however, I have not implemented any skinning or animation.

**Shaders in a graphics application**

Graphics shaders are used to tell the computer how to render each pixel on the screen (Shehata, 2015). Vertex shaders get the vertices of the image and transform them from 3D to 2D, and Pixel shaders process all 2D pixels within 2D polygons that it receives from the vertex shader. A reference/pointer to a shader is needed in C++ to be able to start using shaders in an application. Once a reference has been made to the shader, DirectX11 has to know to use that shader and the textures are sent over as well to the graphics side to be used in the shader (if it is a pixel shader). Once DirectX11 knows what shaders to use, the model can be rendered in the scene. A basic vertex shader will calculate the positions of each vertex in the scene and transform them into 2D coordinates to be passed onto the pixel shader, UV coordinates will also be passed straight through to the pixel shader. A basic Pixel shader will go through every pixel in the scene and calculate the lighting that affects that pixel and outputs the final colour of that pixel to the GPU.

**Basic Requirements:**

* A Floor and a Teapot, One Light should Pulsate on and off and the second light should gradually change colour.
  + All the lighting is done through per-pixel lighting, which is done by calculating each lights’ diffuse light and specular light for the current pixel and adding them together at the end of for the current pixels colour. The two light effects are done in the UpdateScene function in Scene.cpp.
  + The pulsating light effect is done through a sin calculation using the rotate variable in update scene and multiplying the result by 50 to make the lights’ strength fluctuate between 0 and 100 since the result would be between 0 and 2 which is too small to notice.
  + The gradually changing light colour is done by the same calculation, however, a cos calculation is also used. The results are then used to change to colour of the light’s green and blue components. They are then divided by 3 to make sure that the colour changes gradually.
  + The Teapot model can be controlled using I and K to rotate forwards and backwards, J and L to rotate left and right, U and O to rotate around itself and , and . to move backwards and forwards.

* Add a sphere where the vertices ‘wiggle’ and the texture constantly scrolls.
  + For the Wiggling effect, I created a new vertex shader called VertexWiggling\_vs, this shader performs the normal calculations needed in a vertex shader, however the worldPositions coordinates are affected by the sin of the modelPositions opposite coordinate axis + a float variable created in the C++ side that constantly changes, this value is then multiplied by a strength value. The float variable created in the C++ side is updated by a strength value multiplied by the frameTime, in the UpdateScene function. The Wiggle variable is transferred over to the vertex shader through the PerFrameConstants constant buffer.
  + For the Scrolling texture effect, I created a new Pixel shader called TextureScrolling\_ps, this shader returns the colour of the current pixel from the texture sample. The scrolling is done by using the models UV coordinates and multiplying them by the Wiggle variable that was also used in the vertex wiggling shader. The texture sample used for getting the final colour of the pixel will use a sample of the DiffuseSpecularMap at the new UV coordinates. The final colour is then tinted green.
* Add a cube that fades between two textures.
  + I created a new pixel shader called Texture Fading to implement this effect. This shader adds another Texture2D variable in register 2 on the graphics card. All the lighting techniques for the lights in the scene are all the same as the pixel lighting pixel shader. The difference is creating two texture colour variables for the two textures on the cube and getting samples of the two textures colour at the specific UV coordinates for the pixel. A lerp calculation is then performed between the two texture colours and then setting the final Diffuse and Specular Material Colour variables to the result of this calculation.

**Additional tasks**

* Normal mapping and Parallax mapping
  + For normal mapping, I created a new pixel and vertex shader to be used for the models with normal maps. The Normal mapping vertex shader takes in takes in a TangentVertex (new structure that also contains the modelvertex’s tangent) and performs the basic functions of a vertex shader but additionally passing the models Tangent to the pixel shader. The Normal mapping pixel shader has an additional Texture2D variable in register 2 that contains the normal map of the texture that will be passed to it when rendering the model into the scene. The textures normal is calculated using this variable and tangent calculations are performed in the shader. The worldNormal for each pixel is calculated using the texture normal and the tangents to then be added in the light calculations.
  + For Parallax mapping, a new pixel shader is created and used because, the Normal mapping vertex shader can be reused since tangents are needed again. Most of the lighting calculations are the same in the new pixel shader. A new Texture2D variable is used in register 2, for the normal Height Map of the model texture. For the parallax mapping calculations, we get the inverse worldMatrix of the pixel and the direction of the camera using this worldMatrix. The tangentMatrix of the pixel is needed to get the textureOffset direction to be used for the texture height to add to the texture coordinates of the normal Height Map.
* Directional lights
  + For directional lighting, we get the inverse of the normalized light direction and perform a dot product calculation using the world normal of the current pixel to then multiply the result by the lights colour. The specular for the directional light is calculated normalizing the light direction + camera direction to be the halfway. This halfway is then used in a dot product calculation with the world Normal of the current pixel and the result is then raised to the specularPower value and multiplied by the directional light diffuse light. These two values are added with the other lights in the scene to get the final colour of the current pixel in the scene.
* Spotlights
  + For Spotlights, the angle of the spotlight cone is needed to be passed to the GPU from the C++ side to be able to see if the current pixel in the pixel shader is within the spotlights field of view. Every pixel shader that performs lighting calculations had to be changed to accommodate spotlights. If the current pixel that the pixel shader is working on is within the spotlights cone, the spotlights diffuse, and specular lighting are added to the total diffuse and specular lighting values for that pixel. I have implemented Shadow Mapping for the spotlight light calculations.
  + The spotlight can be controlled using T and G to rotate forwards and backwards, F and H to rotate left and right, R and Y to rotate around itself and B and N to move backwards and forwards.
* Blending modes
  + Additive Blending
    - For additive blending, I added a new blendstate in State.cpp, and I set the RenderTargets’ Source and Destination Blends the be BLEND\_ONE with the operation to be used being ADD.
  + Multiplicative Blending
    - For multiplicative blending, I added a new blendstate in State.cpp and I set the RenderTargets’ Source blend to be BLEND\_DEST\_COLOR and I set the Destination blend to BLEND\_ZERO, the operation to be used is still ADD. I then made sure that the models using multiplicative blending were rendered last in the scene.
  + Alpha Blending
    - For Alpha blending, I added a new blendstate in State.cpp and I set the RenderTargets’ Source blend to be the sources alpha and then I set the Destinations blend to be the inverse of the source’s alpha, these are then added together. In the Blending Pixel shader, I performed an alpha check to see if the current pixels alpha value is less than 0.5 and discarded it if true.
* Cell Shading
  + For Cell Shading, the model must be rendered twice, Once for the outline of the model and a second time for the actual model.

For the outline of the model new vertex and pixel shaders were used. The vertex shader gets the view position of the vertex from the cameras viewpoint and expands the world Position of the vertex by the view position and the outline thickness defined from the C++ side. The outline pixel shader returns the outline colour defined from the C++ side for each pixel.

A new pixel shader is created for the Cell Shading effect. This pixel shader has an additional Texture2D variable for the Cell Map and another Sampler State for this Cell Map. For each light calculation, the diffuse level is calculated and then a Cell Diffuse is calculated by getting a sample of the Cell Map at the lights diffuse level and clamping it by the new sampler state. This is done for each light in the scene and then they are added up at the end for the pixels final colour. The clamping of the Cell Diffuse level is done to get small textures to be used in the cell shading effect.

* Improvement of Code to add flexibility.
  + I have made improvements to the code in two ways, the first was through adding additional classes, for the lights and textures that were being used.
    - For the Lights, I added a new CLight class that uses the Model class as a base, where all the user needs to add when implementing the new light is the Mesh, strength, Colour, Position and Scale of the new light. I have also added a SetLightStates Function to the lights where you are able to set the Blend state, Depth state and rasterizer state to be used by the graphics card for the light.
    - For the textures, I added a new CTexture class, that contains a Resource pointer and a shaderResourceView pointer for the texture, you are then able to use the LoadTextureFromHelper function in the class to add a texture to the texture object.
  + The second improvement I made in the code was to add additional functions in the model class to setup all the information the graphics card needs to render a model, such as the pixel or vertex shader, the different states to use and the shader resources.
  + The third and final improvement I made was adding a Light structure in the Common.hlsli and Common.h to be used for the lights in the PerFrameConstants struct whenever a new light was needed to be added to the scene. This Light structure contains the information needed about the light to perform the needed calculations, such as the lights direction, colour and position.

**Improvements or extensions that I could make:**

* One improvement that I could make would be to add shadow mapping for each light in the scene, for example, the directional light that is acting like a sun in the scene.
* Another Improvement I could make would be to slim down the rendering process for each model further.

# References

Shehata, O., 2015. *A Beginner's Guide to Coding Graphics Shaders.* [Online]   
Available at: https://gamedevelopment.tutsplus.com/tutorials/a-beginners-guide-to-coding-graphics-shaders--cms-23313