# UoL logoUniversity of London

# Assessment Coversheet

Complete this coversheet and read the instructions below carefully.

**Candidate Number**:

EX3023

**Degree Title**:

Computer Science

**Course/Module Title**:

3D Graphics and Animation

**Course/Module Code:**

CM3045

**Enter the numbers, and sub-sections, of the questions in the order in which you have attempted them:**

**2a,2b,2c**

**3a,3b**

**Date**: 09 March 2023

**Instructions to Candidates**

1. Complete this coversheet and begin typing your answers on the page below, or, submit the coversheet with your handwritten answers (where handwritten answers are permitted or required as part of your online timed assessment).
2. Clearly state the question number, and any sub-sections, at the beginning of each answer and also note them in the space provided above.
3. For typed answers, use a plain font such as Arial or Calibri and font size 11 or larger.
4. Where permission has been given in advance, handwritten answers (including diagrams or mathematical formulae) must be done on light coloured paper using blue or black ink.
5. Reference your diagrams in your typed answers. Label diagrams clearly.

**The Examiners will attach great importance to legibility, accuracy and clarity of expression.**

**Begin your answers on this page**

**2a.** To implement a fragment shader that draws a colour pattern on an object in unity, first create a project and install the Universal RP and Core RP. After that create a URP renderer in asset folder and set the render pipeline to the new URP renderer. Next create a shader file and include in the Lighting.hlsl and Core.hlsl file from the render-pipelines. Add in \_BaseColor and \_Scale into Properties with a default value of (1,1,1,1) and 10 respectively. Add in uv(TEXCOORD0) and positionsOS(POSITION) in the Attributes and Varyings. Add in positionHCS(SVPOSITION) in Varyings and add in a half4 \_BaseColor, float \_Scale in CBUFFER\_START.In the Varyings vert function, declare Varyings OUT set the OUT.positionHCS to be TransformObjectToHClip(IN.positionOS.xyz). This is to transform the object space to homogenous space. In the vert function, set OUT.uv to be IN.uv, this is to pass in the uv from attribute to the uv varying variable. Next create a half4 frag function which is the fragment shader. Inside the fragment shader, add in an if statement to check if the remainder of IN.uv.y\*\_Scale is bigger than 1.0 but this can return negative value as IN.uv.y so we have to do abs to change it to positive. The if statement will be if(abs(fmod(IN.uv.y\*\_Scale\*intensity,2))>1.0), so if is true, we set the \_BaseColor xyz to be 0(by multiplying it with 0) which will set it as black . Finally, return the colour for the fragment shader. This will give us the white and black stripe. I have used IN.uv.y instead of IN.positionOS.y for the if statement as the lecturer has covered in the later lecture where using uv value will use the coordinates of the pixel rather than using coordinate in the object space which is more accurate as using positionOS.y will have a very large white stripe in the middle of the object where uv.y will give us a uniform black and white stripe on the object. In the inspector, the base color and scale value can be changed so if the base color is orange, it will be orange and black stripe and if the scale value is smaller, there will be lesser stripe vice versa. The way I did this GPU shaders were taught by the Coursera lecture and SIM lecture, so I did not explore other way of implementing this.

**2b.** To animate the white and black stripe, declare a float \_Frequency in Properties and CBUFFER\_START, with a value of 1 in Properties. In the fragment shader, add in the calculation for intensity which is 0.5\*sin(\_Frequency\*\_Time.y)+1.0, this calculation is given by the lecturer and because sin return value of -1 and 1 so adding a 1.0 will rescale the value. The different between using 0.5 of the sin value and using full sin value is that 0.5 sin value will not completely remove the stripe during the animation phase but full sin value will completely remove the stripe during the animation phase.

\_Time.y will return seconds so multiplying it to frequency will increase the value every second which created the animation effect. I used sin because it is taught by the Coursera and SIM lecture but this calculation can be done through cos and tan too. After getting the value from the sin function, multiply it to the uv and scale which will give us the following code IN.uv.y\*\_Scale\*intensity. In unity, there should be animation for the stripe.

**2c.** To create a complex animated shader that includes both vertex definition and fragment based patterns that work together, it is similar concept as the previous task. In the vert function(vertex shader), we can change the shape of an object by using IN.positionOS \*= 0.5\*sin(IN.positionOS)+1.0; What this code means is that sin(IN.positionOS) will return us a value from -1 to 1 and in this case object space position is used as the value to calculate the sin value. Similar to previous, we using half of the sin value so the value will be -0.5 to 0.5 and adding 1 will rescale the value. With the value, multiply it to the object space it will change the object shape. To animate it, we just have to multiply \_Time.y to IN.positionOS so the final code will be IN.positionOS \*= 0.5\*sin(IN.positionOS+\_Time.y)+1.0;. Similarly we can use cos and tan to do the calculation but I used sin because it was taught in Coursera and SIM lecture. So after adding this, in unity we can see the object to be shrinking and growing together with the line animation.

**3a.** The Lambertian component equation of the lighting equation is K­­­­dIi(N·L). Kd is the coefficient of surface material, Ii is the intensity of the light source and (N·L) is the dot product of the surface normal and the light direction. To implement this equation in unity, we must declare \_DiffColor in Properties and CBUFFERSTART. Next we can implement the lighting equation in the half4 frag function(fragment shader). We can find the dot product of N and L which is equivalent to IN.normalWS.xyz and mainLight.direction.xyz respectively. So the full code of dot product of N and L is nl = dot(inp.normalWS.xyz, mainLight.direction.xyz). However, the value return can be negative(the light is not hitting the surface) so we can use max function to remove the negative value so the final code will be nl = max(0, dot(inp.normalWS.xyz, mainLight.direction.xyz)). For Ii, it will be mainLight.color as it is the intensity of the light and Kd will be \_DiffColor as it is the surface color. So if we do

half4 diffuse = \_DiffColor \* mainLight.color \* nl, this will return the Lambertian component equation of the lighting equation.

**3b.** An alternative lighting model equation will be using ambient light + diffuse + specular. The ambient light equation is KaIa where Ka determines how much ambient light is reflected on the object surface and Ia is the ambient light in a scene. The diffuse light equation is K­­­­dIi(N·L). Kd is the coefficient of surface material, Ii is the intensity of the light source and (N·L) is the dot product of the surface normal and the light direction. The specular equation is KsIi(h·n)m where Ks is determine how much specular reflected from the surface, Ii is the intensity of the specular light, and h·nis dot product of h( and n is the surface normal vector. The m variable control the shininess. So, to implement this lighting model into unity, for ambient light we have to declare \_Ambient and \_AmbColor in properties and in CBUFFERSTART. \_Ambient is the intensity which in the equation is Ia and \_AmbColor is the Ka. In fragment shader we can calculate ambient by using amb= \_Ambient \* \_AmbColor. In 3a, the diffuse light equation(Lambertian component equation) already calculated which is K­­­­dIi(N·L) so we can use that value for the full equation later. The specular can be implemented by adding \_Shininess and \_SpecColour into the Properties and CBUFFERSTART. To calculate h we have to get the view direction which is equal to normalizing the world space camera position – world space position so the code will be float3 viewDir = normalize(\_WorldSpaceCameraPos.xyz - inp.positionWS); Then we can find h by h=(viewDir+lightDir)/length(viewDir+lightDir); where lightDir is mainLight.direction.xyz. Next, we can find h·n which is float hdotn = max(0,dot(h,inp.normalWS.xyz)); where n is world space normal. Using max function here is to prevent the value to be negative. Now we can calculate the specular which is float4 spec = pow(hdotn,\_Shininess)\*float4(mainLight.color,1)\*\_SpecColour; where \_Shininess is m, pow is power(exponential) function, float4(mainLight.color,1) is Ii and \_SpecColour is Ks. Now we can implement the full equation which is ambient light + diffuse + specular which will be half4 light = amb+diffuse+spec;