DX1219: PRACTICAL 3

The aim of this practical is to build upon practical 2 and add in our very own lighting.

Normals

1. Based on your current implementation, you should have **color** and **texture** with alpha support.
2. In order to support **lighting**, we need to make use of **normals** which is a 3d vector telling the direction of where the vertex is facing. It is added to vertex data as the normal is send through the vertex like the position and uv. It is then send from the vertex shader to the fragment shader.

```
struct vertexData {
    float4 position : POSITION;
    float2 uv : TEXCOORD0;
    float3 normal : NORMAL;
};

struct vertex2Fragment {
    float4 position: SV_POSITION;
    float2 uv : TEXCOORD0;
    float3 normal : NORMAL;
};
```

3. We set the normal to be the same as from the vertex.

```
vertex2Fragment MyVertexShader(vertexData vd) {
    vertex2Fragment v2f;
    v2f.position = UnityObjectToClipPos(vd.position);
    v2f.uv = TRANSFORM_TEX(vd.uv, _mainTexture);
    v2f.normal = vd.normal;
    return v2f;
}
```

And the change the fragment shader to the following

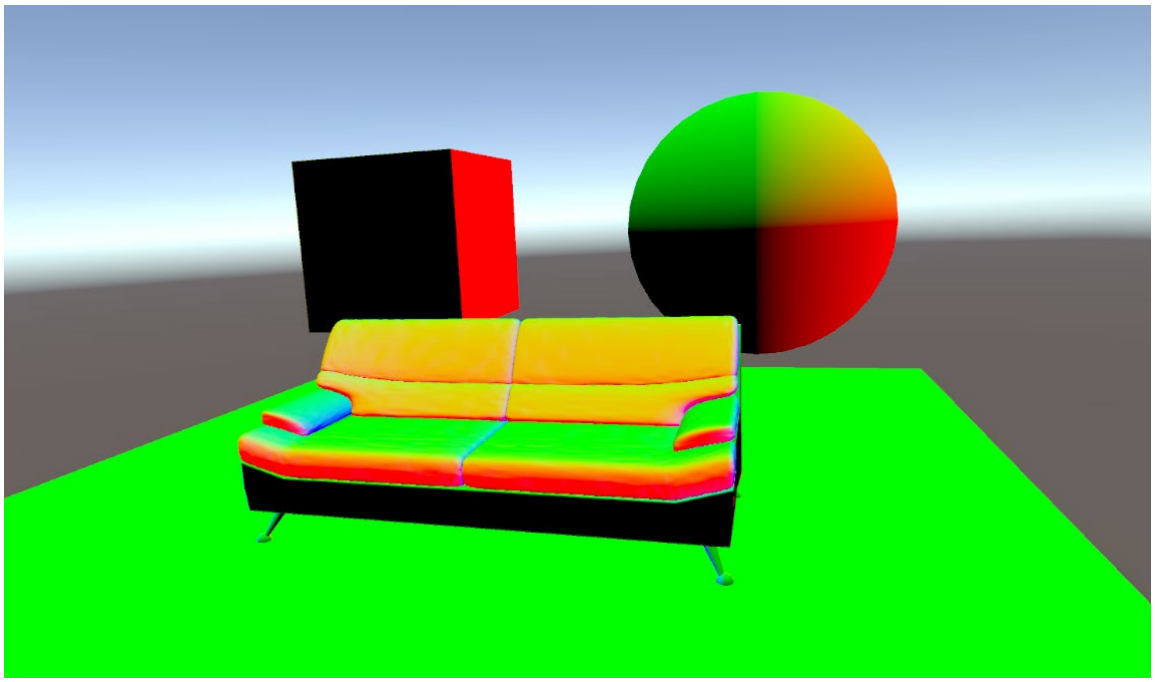
```
float4 MyFragmentShader(vertex2Fragment v2f) : SV_TARGET
{
    float4 result = float4(v2f.normal, 1);
    return result;
}
```

4. I have added a cube and sofa model to the scene as well as lifted the camera and tilt downward. Remember to add the material to all 4-sub model of the sofa. This is how it should look like. Like how we have changed the **color** based on the UV in the first practical so this time we are coloring based on the value of the **Normal** of each fragment.

5. Now let add in the following rotating script in Unity to all 4 objects. (Not a shader)

```
public float rotatingSpeed = 20;
// Start is called before the first frame update
Unity Message | – references
void Start()...

// Update is called once per frame
Unity Message | – references
void Update()
{
    transform.Rotate(new Vector3(0, rotatingSpeed, 0) * Time.deltaTime);
}
```

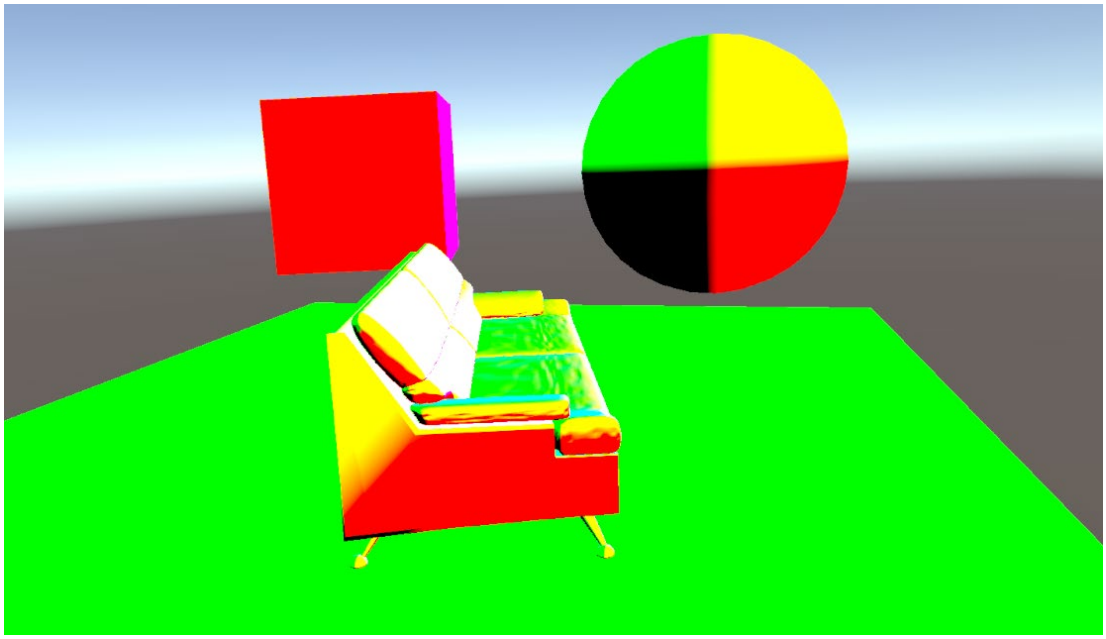


6. As you rotate the object, you realise that the color stays the same. You should be thinking that the color should be changing as it rotates as the normal should be changing. Think of an object in the light, as the object rotate, the shading will change accordingly. This is because

we using the normal in **local space**. What we need is the normal in **world space.** Do this in the vertex shader.

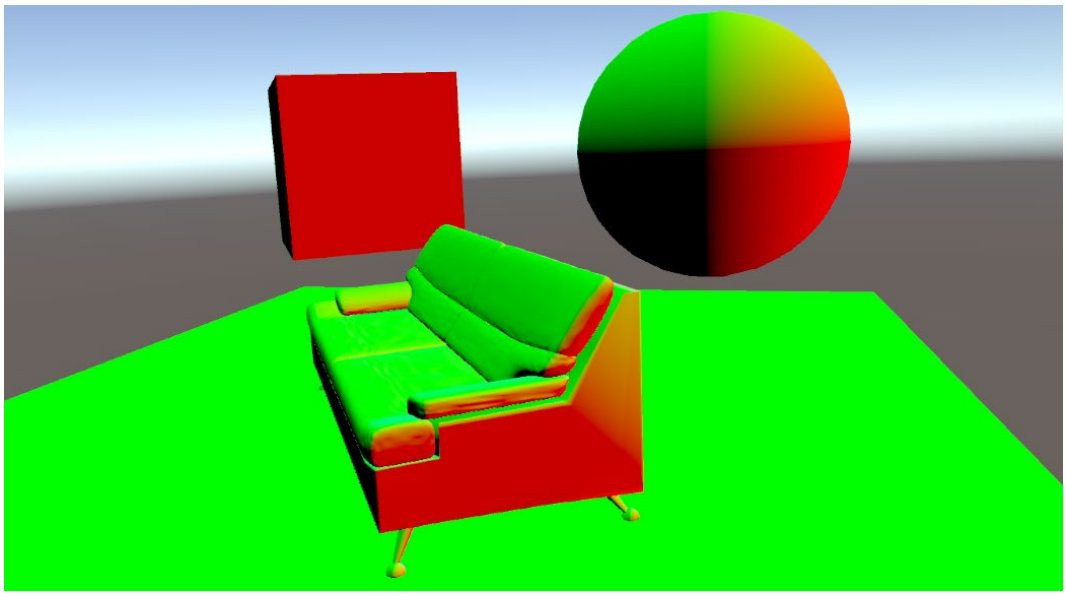
```
v2f.normal = mul(unity_ObjectToWorld,float4(vd.normal,0));
```

unity_ObjectToWorld is the matrix that bring the vertex **from object/local space to world space**.

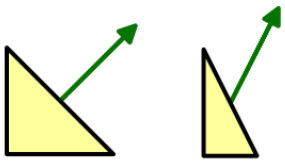


7. This look better and the shading will look a bit strange as it is too bright. This is due to the fact that the normal after the world to object transform might have a magnitude bigger than one so the object look very bright. Therefore, we just normalize the normal vector to have a magnitude of 1 only. Feel free to combine them together to 1 line.

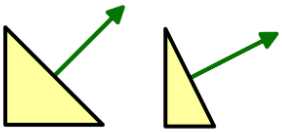
```
v2f.normal = mul(unity_ObjectToWorld, float4(vd.normal, 0));
v2f.normal = normalize(v2f.normal);
```

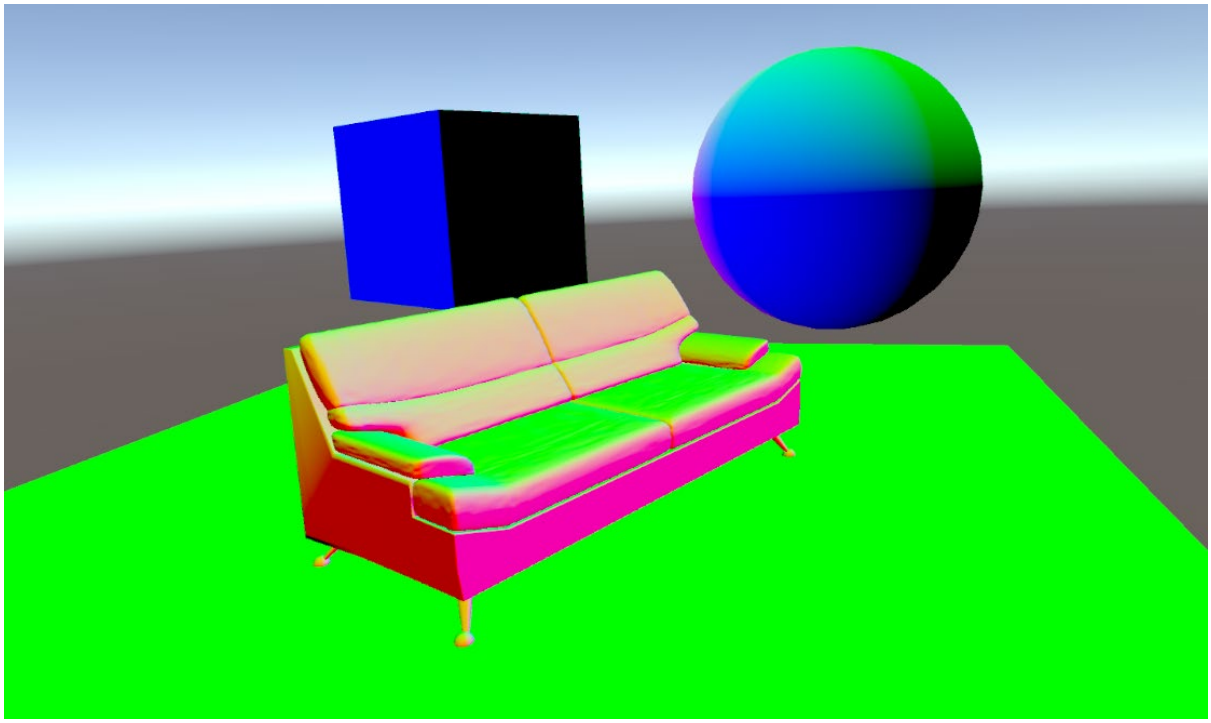


8. When we use the current method, it will have one issue. While we are using normalized vectors. Objects that are not uniformly scaled will be wrong.



We need transpose the world to object matrix and multiply the result with the vertex normal. The transpose





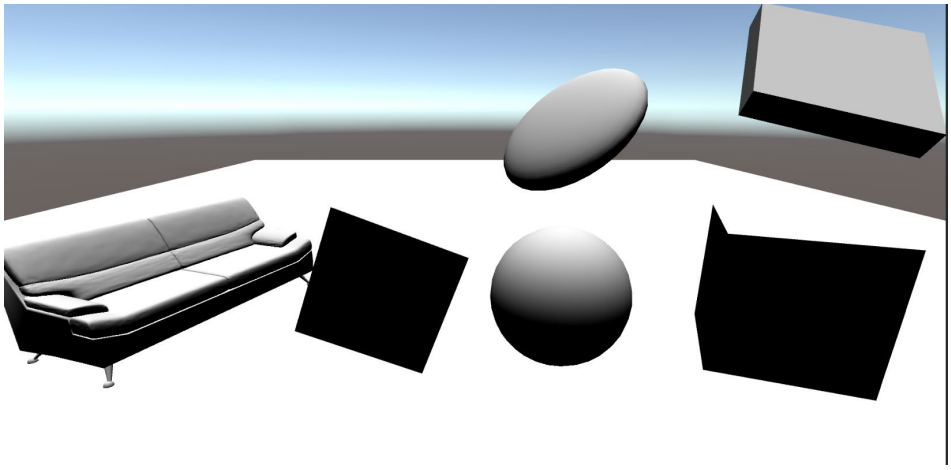
```
v2f.normal = mul(transpose((float3x3)unity_WorldToObject),float4(vd.normal,0));
v2f.normal = normalize(v2f.normal);
```

9. We can use a unity function that help us do that part. It uses explicit matrix multiplication so it will be faster than the transpose code.

```
v2f.normal = UnityObjectToWorldNormal(vd.normal);
v2f.normal = normalize(v2f.normal);
```

10. Let try doing some lighting now. Object reflect light and hit our eyes. Imagine the sun shining down from above.

```
float diffuse = dot(float3(0,1,0), v2f.normal);
float4 result = float4(diffuse, diffuse, diffuse, 1.0);
return result;
```



```
float diffuse = saturate(dot(float3(0,1,0), v2f.normal));
```

https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-saturate

11. Clamp between 0 and 1 using the saturate function. We don't want negative light where the light direction and normal is larger than 90 degrees where its cosine is negative.

Diffuse Shading

12. Let try create our very own light source instead of using Unity's directional light that was created by default. The ExecuteInEditMode keyword refers to the game object updating even in Edit Mode.

```
[ExecuteInEditMode]
public class LightObject : MonoBehaviour
{

}
```

Add the following variables. We will set the default direction to be point down with (0, -1, 0). We will create the getter and for Direction and Position as well.

```
[SerializeField]
private Vector3 direction = new Vector3(0, -1, 0);
```

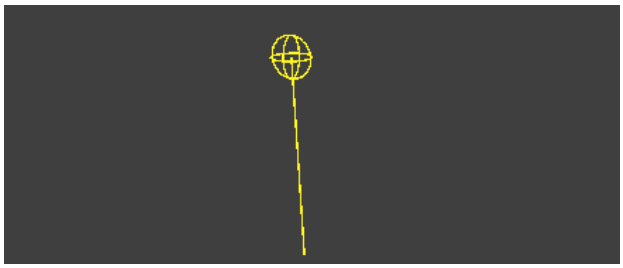
13. Set the **Update** function to update the direction based on the rotation value in the game object's transform. We will normalized it as well.

```
// Update is called once per frame
void Update()
{
    direction = transform.rotation * new Vector3(0, -1, 0);
    direction = direction.normalized;
}
```

14. In order to draw it only inside the editor mode, we will use **OnDrawGizmos** to draw the position and direction using a wire sphere and ray.

```
private void OnDrawGizmos()
{
    // Draw a yellow sphere at the transform's position
    Gizmos.color = Color.yellow;
    Gizmos.DrawWireSphere(transform.position, 1);
    Gizmos.DrawRay(transform.position, direction * 10.0f);
}
```

You should see something like that by default to simulate the light source with direction.



15. Let update our shader to support this.

Properties

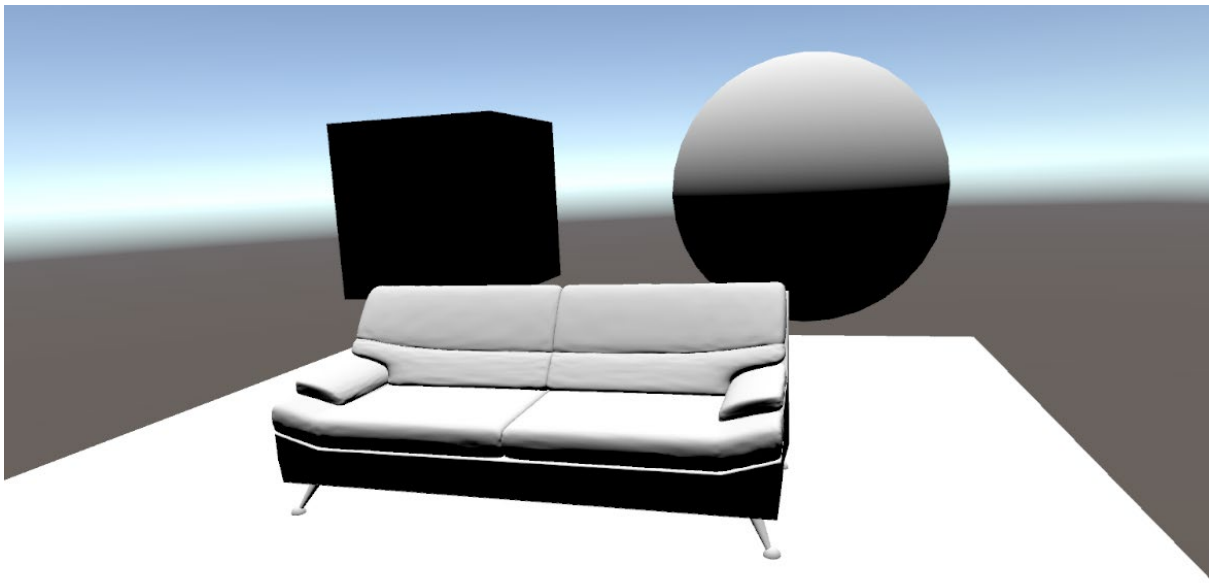
```
_lightPosition("Light Position", Vector) = (0,0,0)
_lightDirection("Light Direction", Vector) = (0,-1,0)
```

Shaders

```
//Light Data
uniform float3 _lightPosition;
uniform float3 _lightDirection;
```

```
float4 MyFragmentShader(vertex2Fragment v2f) : SV_TARGET
{
    float diffuse = saturate(dot(-_lightDirection, v2f.normal));
    float4 result = float4(diffuse,diffuse,diffuse, 1.0);
    return result;
}
```

You should see something like this. With the default value we provide. How do we send data over to the Shader from Unity?



16. Let update our LightObject script to support sending to the shader. First, let add a material variable to update. Next, let write a function to send data from the material to the shader. Call that function in the Update function.

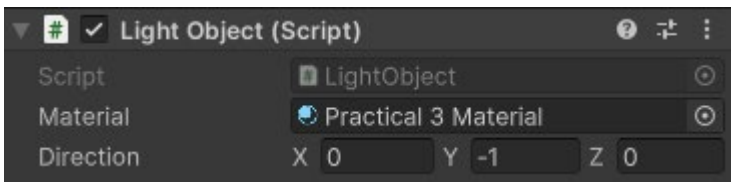
```
[SerializeField]
private Material material;
```

```
private void SendToShader()
{
    material.SetVector("_lightPosition", transform.position);
    material.SetVector("_lightDirection", direction);
}
```

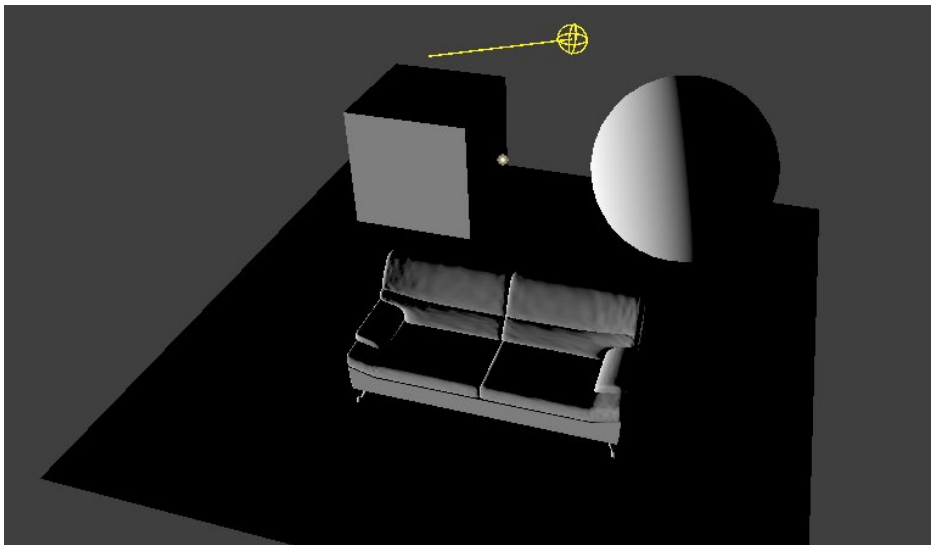
```
// Update is called once per frame
void Update()
{
    direction = transform.rotation * new Vector3(0, -1, 0);
    direction = direction.normalized;

    SendToShader();
}
```

Remember to add the material to the Light Object Script in the inspector.



Trying changing the Rotation value in the transform and you should see something like this where the shading changes.



17. Let add color to our lights. Let update the shader to support that.

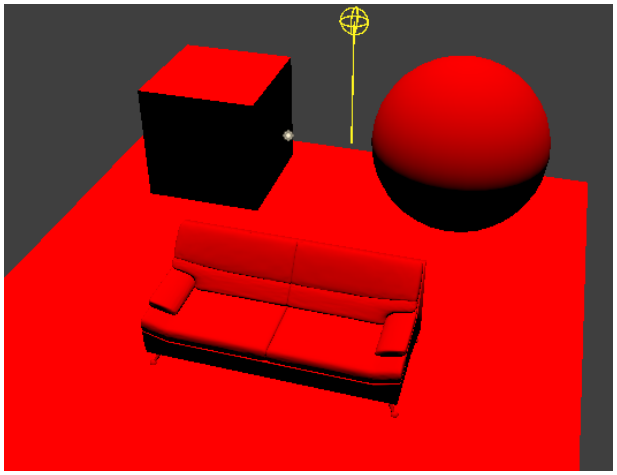
```
_lightPosition("Light Position", Vector) = (0,0,0)
_lightDirection("Light Direction", Vector) = (0,-1,0)
_lightColor("Light Color", Color) = (1,1,1,1)
```

```
//Light Data
uniform float3 _lightPosition;
uniform float3 _lightDirection;
uniform float4 _lightColor;
```

Next, we will update the fragment shader.

```
float4 MyFragmentShader(vertex2Fragment v2f) : SV_TARGET
{
    float3 diffuse = _lightColor * saturate(dot(-_lightDirection, v2f.normal));
    float4 result = float4(diffuse, 1.0);
    return result;
}
```

Try change the color value in the material and you should see something like that.



18. Let add the color support the LightObject script as well. We will add in a new variable and update the **SendToShader** function.

```
[SerializeField]
private Color lightColor;
```

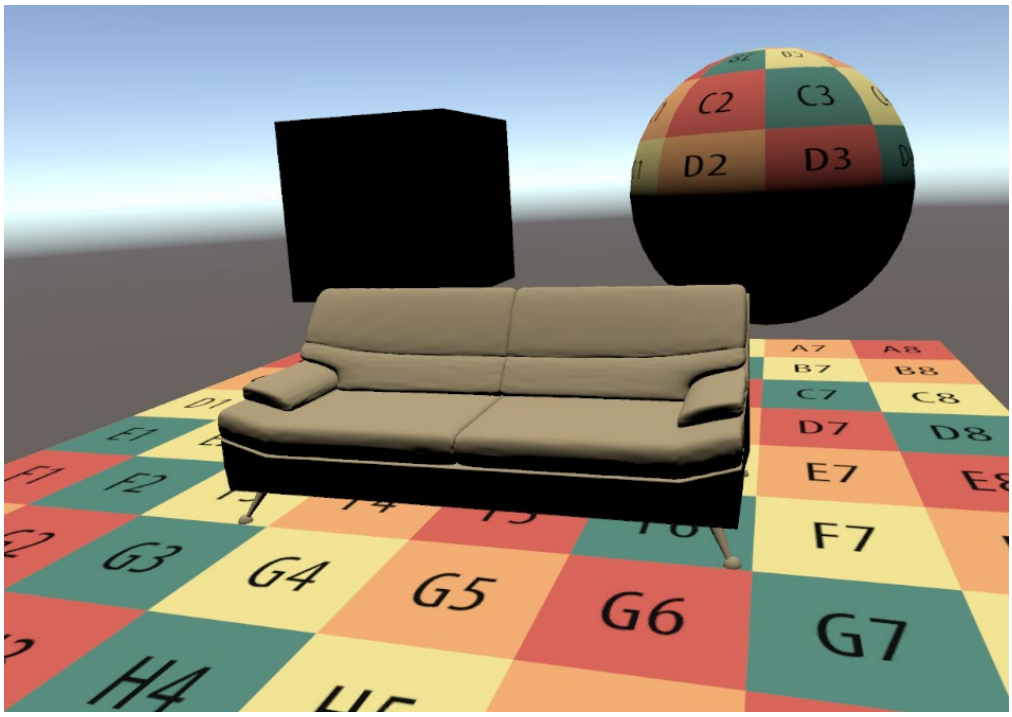
```
private void SendToShader()
{
    material.SetVector("_lightPosition", transform.position);
    material.SetVector("_lightDirection", direction);
    material.SetColor("_lightColor", lightColor);
}
```

19. Let add in the texture support from the previous practical. We usually use the term Albedo for the base color / texture in lighting. Albedo refers to the expression of the ability of surfaces to reflect sunlight. We will make use of the texture and tint and combine it with the light color in our fragment shader.

```
Properties{
        _tint("Tint", Color) = (1, 1, 1, 1)
        _mainTexture("Albedo", 2D) = "white" {}
```

```
float4 MyFragmentShader(vertex2Fragment v2f) : SV_TARGET
{
    float4 albedo = tex2D(_mainTexture, v2f.uv) * _tint;
    float3 diffuse = albedo.xyz * _lightColor * saturate(dot(-_lightDirection, v2f.normal));
    float4 result = float4(diffuse, albedo.w);
    return result;
}
```

You should see something like that with white light and the grid texture selected.



Specular Shading

20. Beside diffuse reflections, there are also specular reflections. When light doesn't get diffused after hitting a surface, instead, the light ray bounces off the surface at the angle equal to the angle hitting the surface. It is what cause the reflections in the mirror.
    In diffuse lighting, we do not care about the position of the viewer / camera but in specular shading we do. Imagine looking at a metal pot while moving your head, the shiny spot will update according to where our eyes are. Therefore, we need to send the **world position** from the vertex shader to the fragment shader.

```
struct vertex2Fragment {
    float4 position: SV_POSITION;
    float2 uv : TEXCOORD0;
    float3 normal : NORMAL;
    float3 worldPosition : POSITION1;
};
```

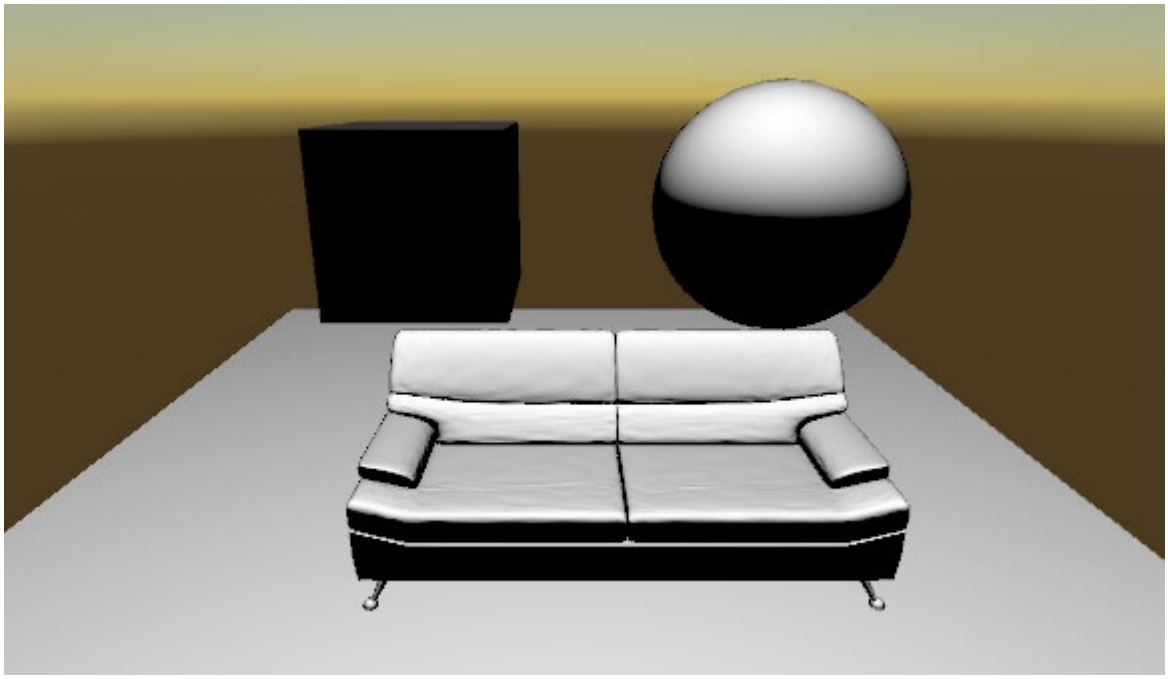
21. In the **vertex shader**, we multiply the local position using the object to world matrix provided in unity. The matrix is **unity_ObjectToWorld**.

```
vertex2Fragment v2f;
v2f.position = UnityObjectToClipPos(vd.position);
v2f.worldPosition = mul(unity_ObjectToWorld, vd.position);
```

22. In the **fragment shader**, we get the view direction from subtracting the camera position to the world position. Next, we grab the reflection direction vector using the negative light direction reflecting against the normal. We can use the **reflect** function provided in hlsl. Lasting we get the dot product of the view direction against the reflection direction. Let view only the specular for now. Comment some of the previous codes, we will use some of them later. *(_WorldSpaceCameraPos is a Unity keyword to access the Camera Pos in world space)*

```
    float4 albedo = tex2D(_mainTexture, v2f.uv) * _tint;
    float3 viewDirection = normalize(_WorldSpaceCameraPos - v2f.worldPosition);
    float3 reflectionDirection = reflect(-_lightDirection, v2f.normal);
    float result = float(saturate(dot(-viewDirection, reflectionDirection)));
    return float4(result, result, result, 1.0f);
```

You can try moving the light's position and direction as well.



You should have something like this to see the specular shading.

23. The size of the highlight produced depends on the roughness of the material. Smooth materials focus the light better. We can control this smoothness using the material. Let add this to the material properties as well as the uniform.

```
    _lightColor("Light Color", Color) = (1,1,1,1)
    _smoothness("Smoothness", Range(0, 1)) = 0.5
}
```

```
uniform float4 _lightColor;
uniform float _smoothness;
```

In order to make use of smoothness more visible. We raise the reflection by 100 times the power of the smoothness and you should have something more similar to what you expect out of a specular effect.

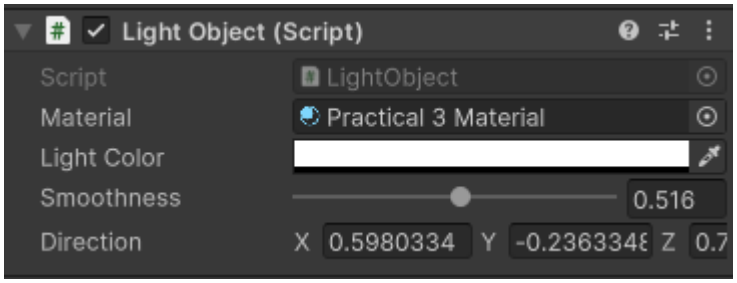
```
float3 reflectionDirection = reflect(-_lightDirection, v2f.normal);
float result = pow(float(saturate(dot(-viewDirection, reflectionDirection))), _smoothness * 100);
return float4(result, result, result, 1.0f);
```

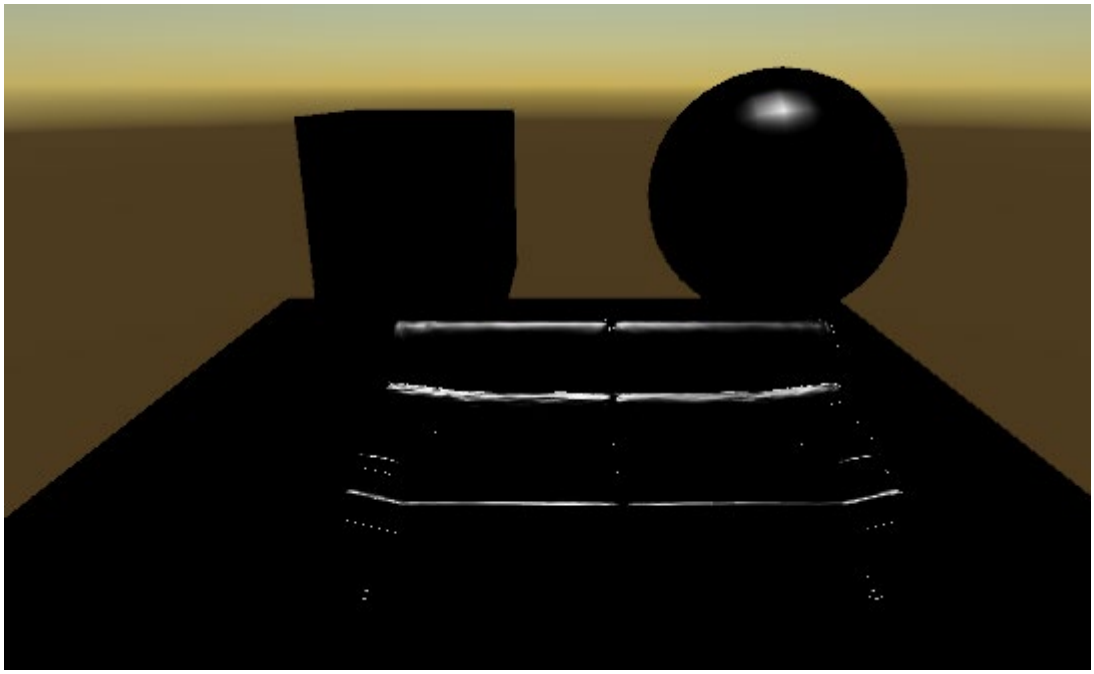
Trying change the smoothness to check. Let add smoothness to the LightObject as well.

```
[SerializeField]
[Range(0f, 1f)]
private float smoothness;
```

```
private void SendToShader()
{
    material.SetVector("_lightPosition", transform.position);
    material.SetVector("_lightDirection", direction);
    material.SetColor("_lightColor", lightColor);
    material.SetFloat("_smoothness", smoothness);
}
```

The light object should look something like that.

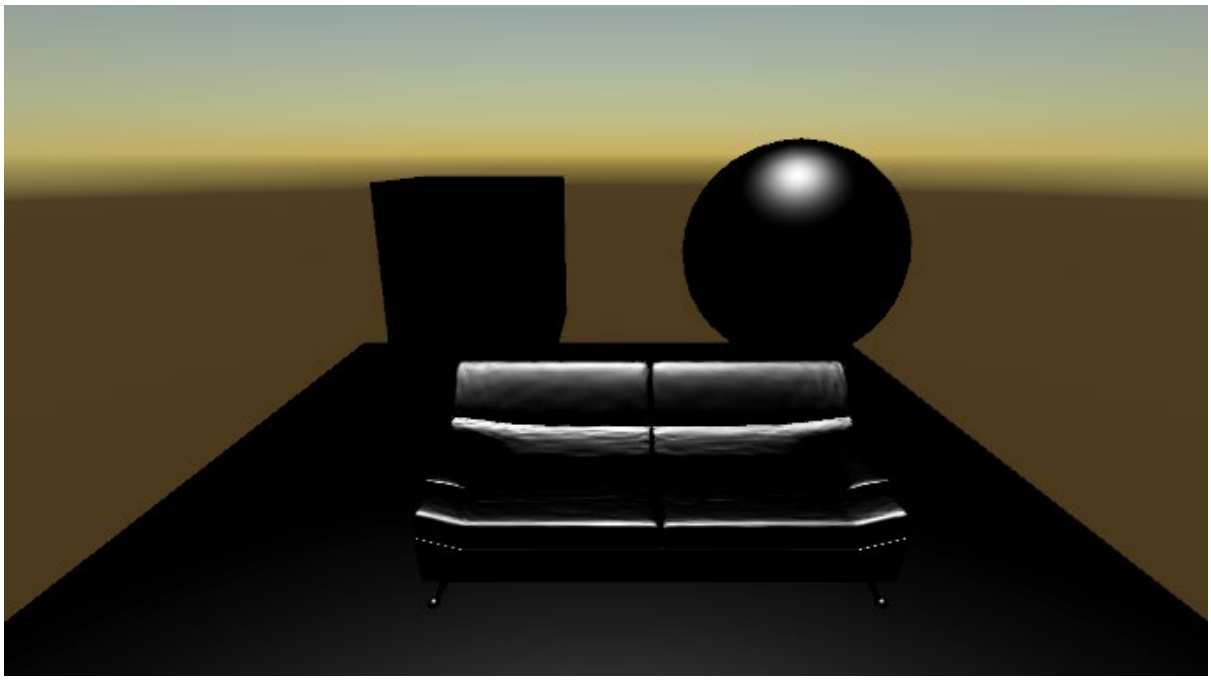




24. What we been using the Blinn reflection model. The most common used model is a improvement to the Blinn reflection model known as the Blinn-Phong. It makes use of a vector halfway between the light direction and view direction. The dot product between the normal and half vector determines the specular shading. Let us also normalize the normal in fragment shader to ensure it is more smooth to prevent tile effects seen on the sphere.

```
float4 MyFragmentShader(vertex2Fragment v2f) : SV_TARGET
{
    v2f.normal = normalize(v2f.normal);
    float4 albedo = tex2D(_mainTexture, v2f.uv) * _tint;
    float3 viewDirection = normalize(_WorldSpaceCameraPos - v2f.worldPosition);
    float3 reflectionDirection = reflect(-_lightDirection, v2f.normal);
    float3 halfVector = normalize(viewDirection + -_lightDirection);
    float specular = pow(float(saturate(dot( v2f.normal, halfVector))), _smoothness * 100);
    float3 specularColor = specular * _specularStrength * _lightColor.rgb ;
    return float4(specularColor, albedo.a);
```

Try playing around with the position and direction.



Let add specular strength to the shader so that it does not have a white hue if it is not a reflective object.

```
_specularStrength("Specular Strength", Range(0, 1)) = 0.5
_smoothness("Smoothness", Range(0, 1)) = 0.5
```

```
uniform float _specularStrength;
uniform float _smoothness;
```

```
float3 specularColor = specular * _specularStrength * _lightColor.rgb ;
return float4(specularColor, albedo.a);
```

Play around with the value of the strength and add it to the **LightObject** script yourself. You should know how to do it by now.

25. Let combine everything together by making use of the albedo (texture), the specular color, the diffuse color to form our Blinn-Phong Lighting.

```
float4 MyFragmentShader(vertex2Fragment v2f) : SV_TARGET
{
    v2f.normal = normalize(v2f.normal);
    float4 albedo = tex2D(_mainTexture, v2f.uv) * _tint;
    float3 viewDirection = normalize(_WorldSpaceCameraPos - v2f.worldPosition);
    float3 reflectionDirection = reflect(-_lightDirection, v2f.normal);
    float3 halfVector = normalize(viewDirection + -_lightDirection);
    float specular = pow(float(saturate(dot( v2f.normal, halfVector))), _smoothness * 100);
    float3 specularColor = specular * _specularStrength * _lightColor.rgb ;
    float3 diffuse = albedo.rgb * _lightColor.rgb * saturate(dot( v2f.normal, -_lightDirection));
    float3 finalColor = diffuse + specularColor;

    // Return the final color with alpha from the albedo texture.
    return float4(finalColor, albedo.a);
}
```