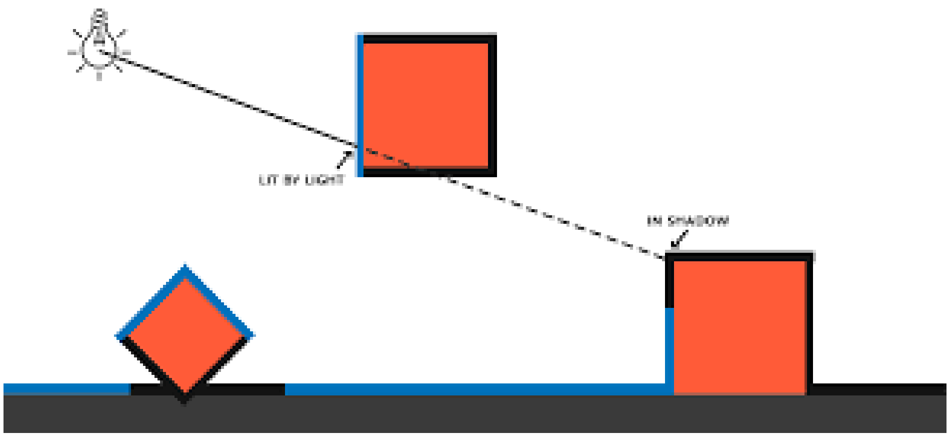
DX1219: PRACTICAL 5

The aim of this practical is to learn how to generate a basic hard shadow using our own shadow map renderer and shader.

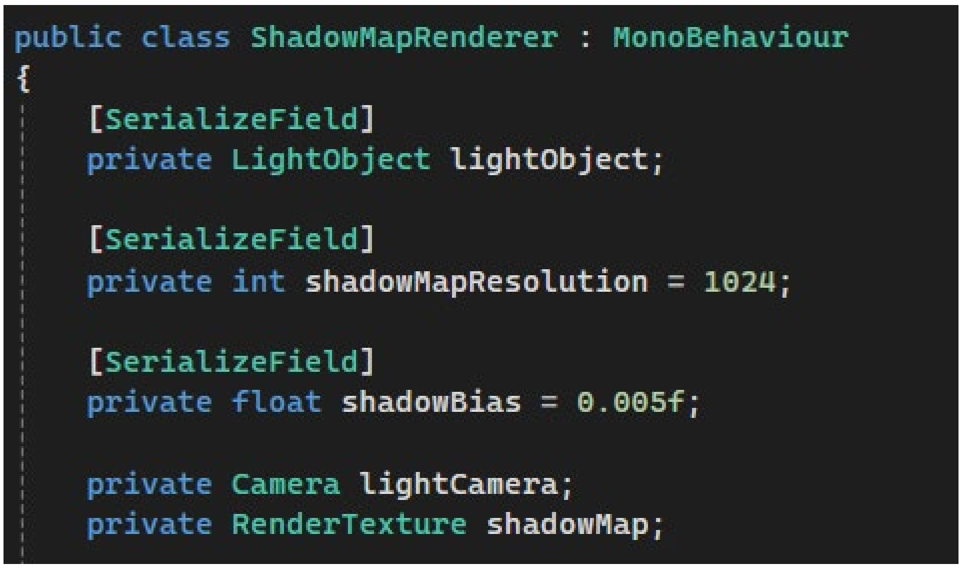
Light Camera Setup



As we have to “view” from the light perspective, we have to create a camera based on the light source which is our Light Object. First, create a script named **ShadowMapRenderer.**

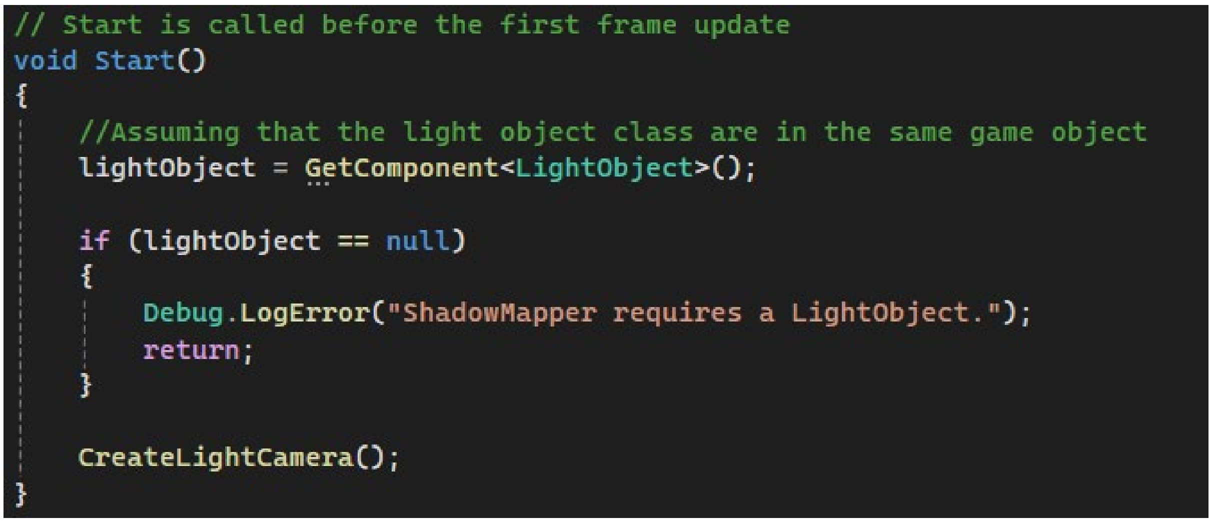
Here is the class with its variables. We will make use of the **LightObject** class. **shadowMapResolution** will decide the resolution of the texture of the shadow map.

shadowBias will be used to help with shadow acne, we increase and decrease as needed. **lightCamera** and **shadowMap** will be camera and texture that will be created by this script.



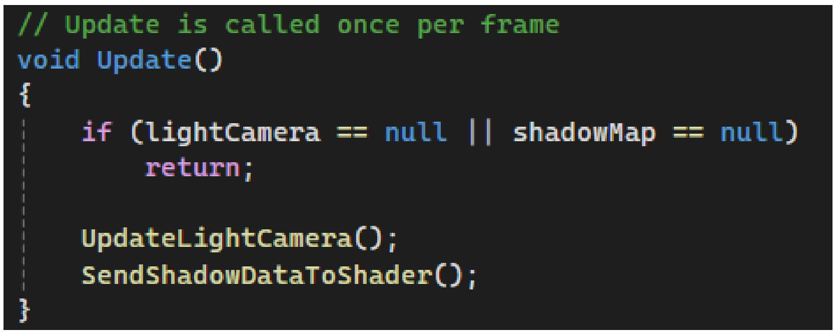
```
public class ShadowMapRenderer : MonoBehaviour
{
    [SerializeField]
    private LightObject lightObject;

    [SerializeField]
    private int shadowMapResolution = 1024;

    [SerializeField]
    private float shadowBias = 0.005f;

    private Camera lightCamera;
    private RenderTexture shadowMap;
```

This is the Start function. It requires the **LightObject** game object. It would find it if we put both the **ShadowMapRenderer** class and LightObject **class** in the same game object. Do not worry about the function that we have not created yet.



```
// Start is called before the first frame update
void Start()
{
    //Assuming that the light object class are in the same game object
    lightObject = GetComponent<LightObject>();

    if (lightObject == null)
    {
        Debug.LogError("ShadowMapper requires a LightObject.");
        return;
    }

    CreateLightCamera();
}
```

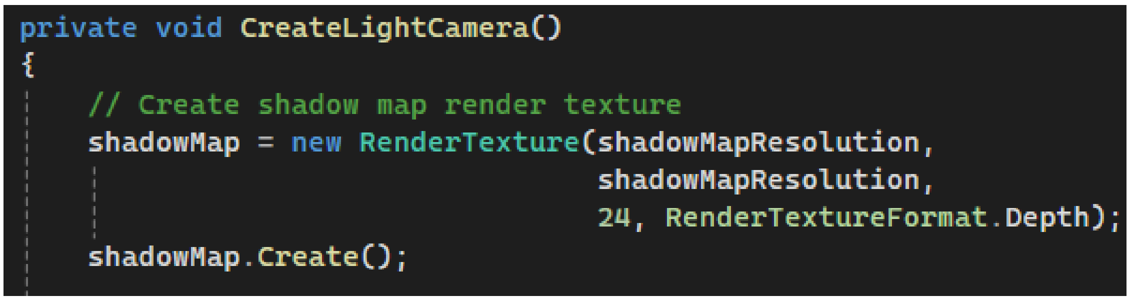
Next, let create the Update function. If the lightCamera is not created or shadowMap has not been created we will not run the update function.



```
// Update is called once per frame
void Update()
{
    if (lightCamera == null || shadowMap == null)
        return;

    UpdateLightCamera();
    SendShadowDataToShader();
}
```

Creating The Light Camera

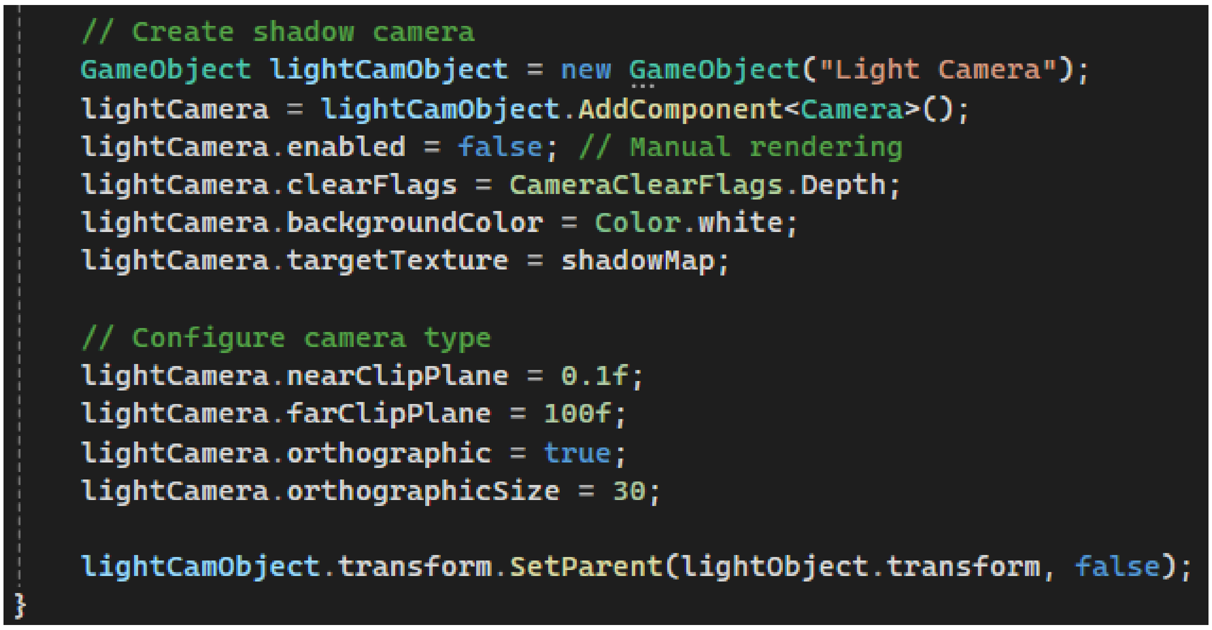
Next, let create the Light Camera function. We will create the light camera in run time. So if we notice that the **lightCamera** object has been created before we destroy it and start over. We will create the shadow map texture by create a new **RenderTexture** that has 1024 x 1024 by default and 24 depth bits. We set it to be a depth render texture format.



```
private void CreateLightCamera()
{
    // Create shadow map render texture
    shadowMap = new RenderTexture(shadowMapResolution,
                                  shadowMapResolution,
                                  24, RenderTextureFormat.Depth);
    shadowMap.Create();
```

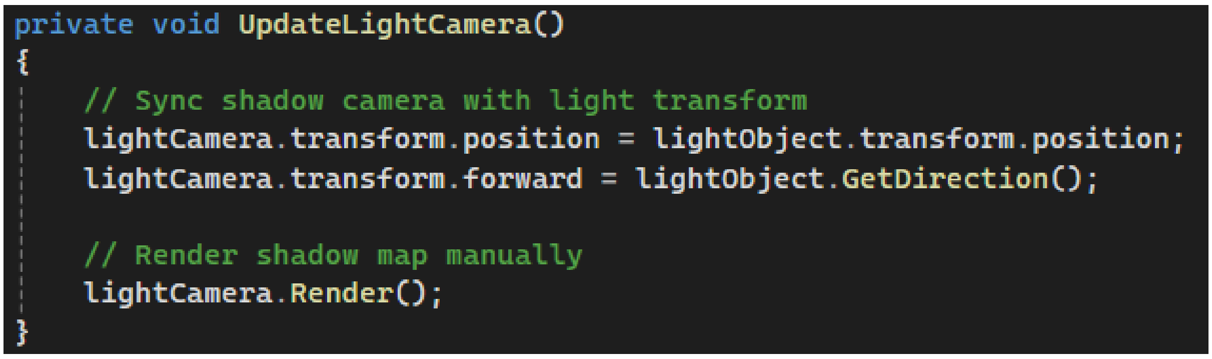
We will create a new object and add the Camera component to it. We will set the enabled to be false as we will be doing Manual rendering. Remember to set the **targetTexture** to be **shadowMap render texture** that we just created. This will cause the camera to draw to the texture instead. You can use this same technique to create mirror, security cameras or even portal effects.

We will also set the **lightCamera** properties. We will set the near clip plane and far clip plane to be between **0.1f** and **100.0f**. This will be what the camera can see. You might need to change these values to fit your scene. Next, we will set the camera to be **orthographic** and set the size to be **30.0** by default. You might need a bigger size for a bigger scene. Remember to set the parent to be on the light object itself.



```
    // Create shadow camera
    GameObject lightCamObject = new GameObject("Light Camera");
    lightCamera = lightCamObject.AddComponent<Camera>();
    lightCamera.enabled = false; // Manual rendering
    lightCamera.clearFlags = CameraClearFlags.Depth;
    lightCamera.backgroundColor = Color.white;
    lightCamera.targetTexture = shadowMap;

    // Configure camera type
    lightCamera.nearClipPlane = 0.1f;
    lightCamera.farClipPlane = 100f;
    lightCamera.orthographic = true;
    lightCamera.orthographicSize = 30;

    lightCamObject.transform.SetParent(lightObject.transform, false);
}
```

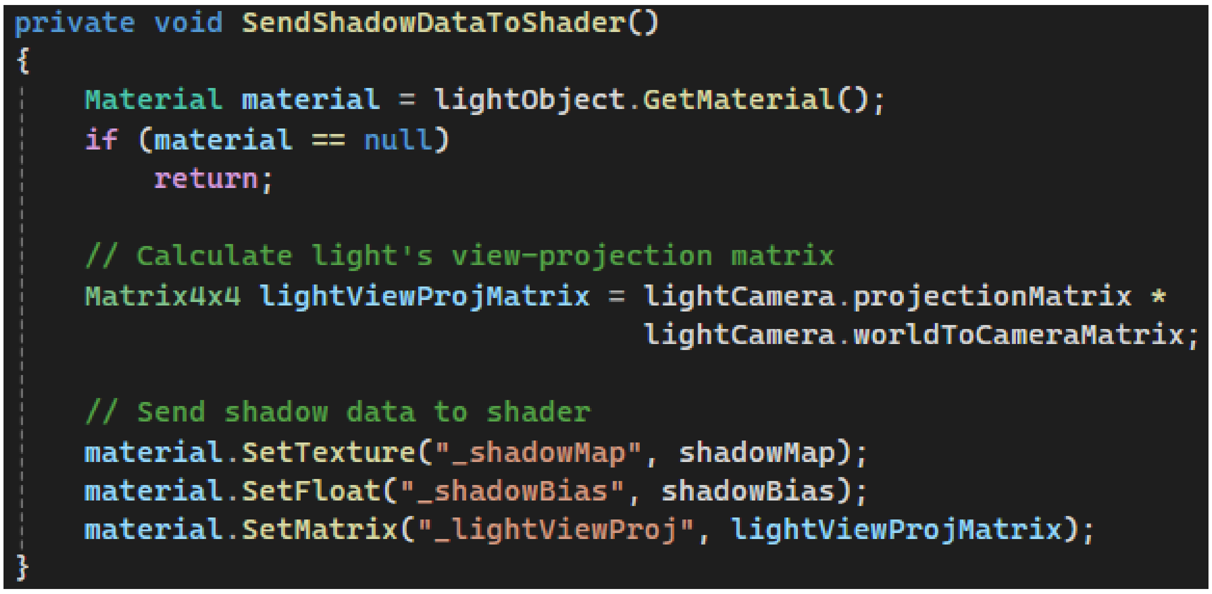
Let create the **UpdateLightCamera** function will be constantly called in the **Update** function. We set the **lightCamera** to be the same position in the **lightObject**. Being the parent, it should already follow the **lightObject**. Next, we want the light camera's forward which is where the camera is facing to be the same direction as the **lightObject**. The **lightCamera.Render** function is to let the light camera be render manually and draw the shadow map texture.



```
private void UpdateLightCamera()
{
    // Sync shadow camera with light transform
    lightCamera.transform.position = lightObject.transform.position;
    lightCamera.transform.forward = lightObject.GetDirection();

    // Render shadow map manually
    lightCamera.Render();
}
```

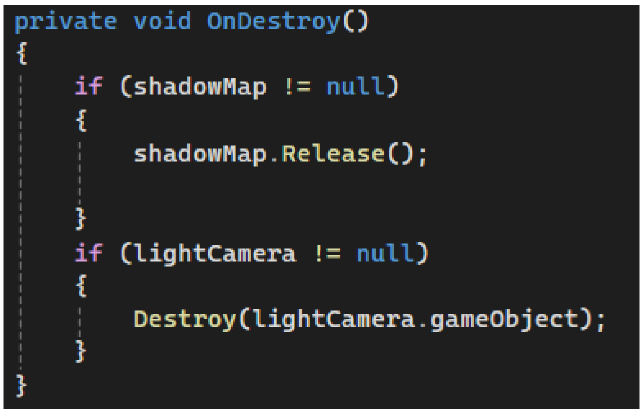
Sending Data to the Shader

We will make sure we can get the material from the light object class object. Next, we have to calculate the **light camera view project matrix**. Remember, this is different from the MVP we been using in the past as that is from the main camera. It is from the perspective of the light camera. We go from **World Space** to **Light Camera Space** to **Clip Space.** We will also update the shadow map texture and shadow bias if changed.



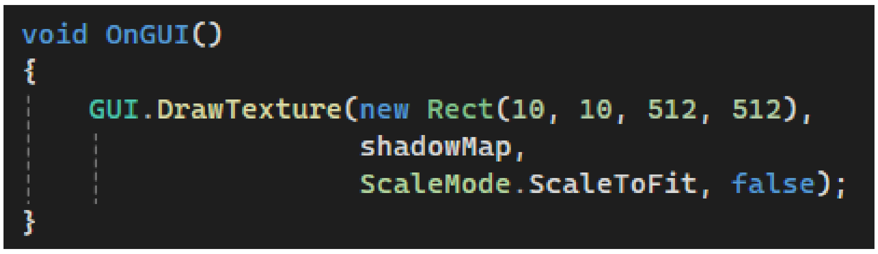
```
private void SendShadowDataToShader()
{
    Material material = lightObject.GetMaterial();
    if (material == null)
        return;

    // Calculate light's view-projection matrix
    Matrix4x4 lightViewProjMatrix = lightCamera.projectionMatrix *
                                    lightCamera.worldToCameraMatrix;

    // Send shadow data to shader
    material.SetTexture("_shadowMap", shadowMap);
    material.SetFloat("_shadowBias", shadowBias);
    material.SetMatrix("_lightViewProj", lightViewProjMatrix);
}
```

We will release the texture when the object is destroyed. Also, delete the camera so that it does not multiple.



```
private void OnDestroy()
{
    if (shadowMap != null)
    {
        shadowMap.Release();

    }
    if (lightCamera != null)
    {
        Destroy(lightCamera.gameObject);
    }
}
```

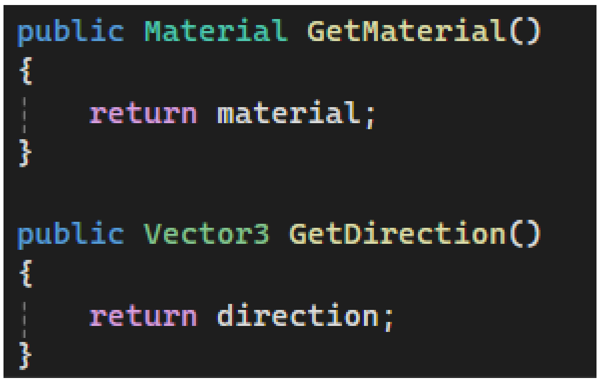
We will draw a Texture to assist us with our debugging. You can remove this when you have done debugging.



```
void OnGUI()
{
    GUI.DrawTexture(new Rect(10, 10, 512, 512),
                    shadowMap,
                    ScaleMode.ScaleToFit, false);
}
```

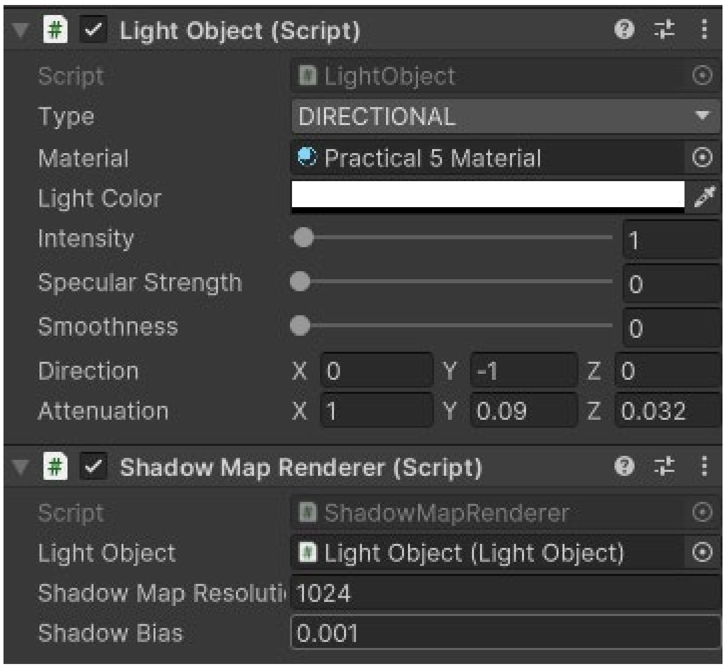
Update Light Object Class

Update the **LightObject** class with these 2 functions that are used in **ShadowMapRenderer** class



```
public Material GetMaterial()
{
    return material;
}

public Vector3 GetDirection()
{
    return direction;
}
```

Add the **ShadowMapRenderer** class to the **LightObject** game object that you have created.



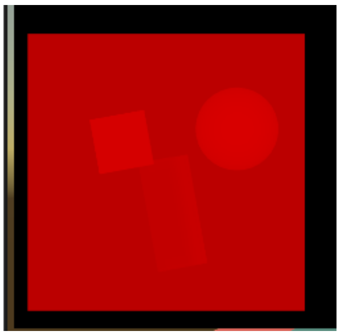
Light Object (Script)

Script	LightObject
Type	DIRECTIONAL
Material	Practical 5 Material
Light Color	
Intensity	1
Specular Strength	0
Smoothness	0
Direction	X 0 Y -1 Z 0
Attenuation	X 1 Y 0.09 Z 0.032

Shadow Map Renderer (Script)

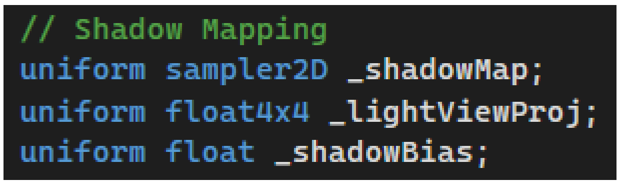
Script	ShadowMapRenderer
Light Object	Light Object (Light Object)
Shadow Map Resoluti	1024
Shadow Bias	0.001

You should see something like that at the top left side. The red is the depth texture. The lighter red are pixels that are closer to the light camera. Darker red are further away. Blacks are empty pixel meaning no objects are within the camera near and far.



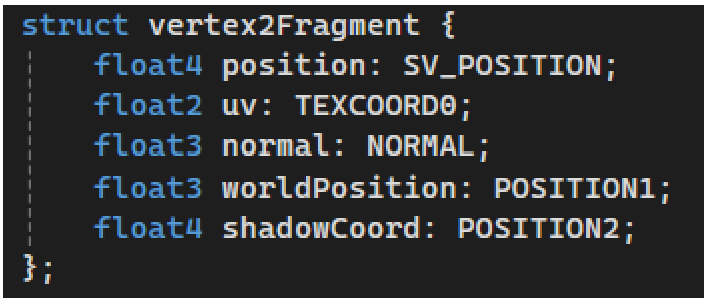
Update Shader To Support Shadow Mapping

These are the 3 variables that we are sending from the application to the shader. **_shadowMap** is the texture, **_lightViewProj** is the projection matrix that bring the point from world space to light clip space. **_shadowBias** is used to remove any shadow ance



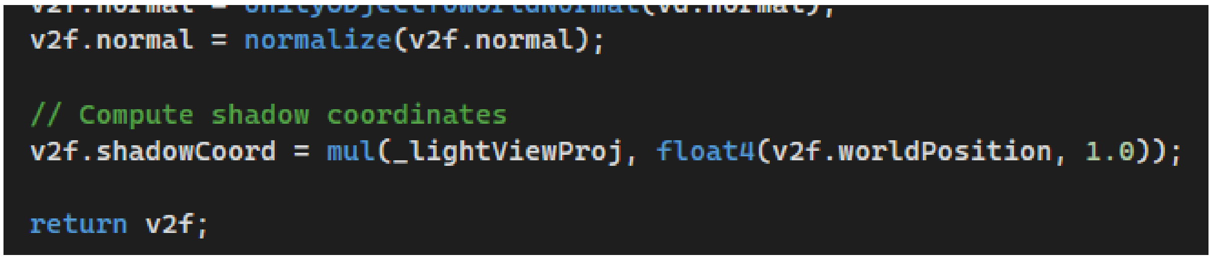
```
// Shadow Mapping
uniform sampler2D _shadowMap;
uniform float4x4 _lightViewProj;
uniform float _shadowBias;
```

In our vertex2Fragment struct, we add in another **float4** named **shadowCoord**. It will be used as the shadow coordinates for getting the shadow texture map.



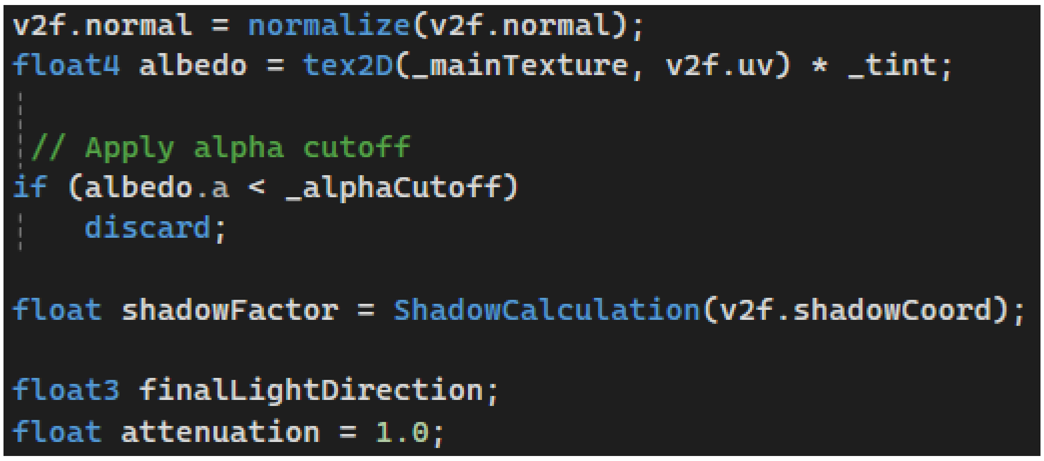
```
struct vertex2Fragment {
    float4 position: SV_POSITION;
    float2 uv: TEXCOORD0;
    float3 normal: NORMAL;
    float3 worldPosition: POSITION1;
    float4 shadowCoord: POSITION2;
};
```

In the vertex shader, calculate the point by bring the vertex point to light clip space by using the projection matrix.



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

// Compute shadow coordinates
v2f.shadowCoord = mul(_lightViewProj, float4(v2f.worldPosition, 1.0));

return v2f;
```

In the fragment shader, we will add this line to calculate the shadow factor.



```
v2f.normal = normalize(v2f.normal);
float4 albedo = tex2D(_mainTexture, v2f.uv) * _tint;

// Apply alpha cutoff
if (albedo.a < _alphaCutoff)
    discard;

float shadowFactor = ShadowCalculation(v2f.shadowCoord);

float3 finalLightDirection;
float attenuation = 1.0;
```

Calculating The Shadow Factor

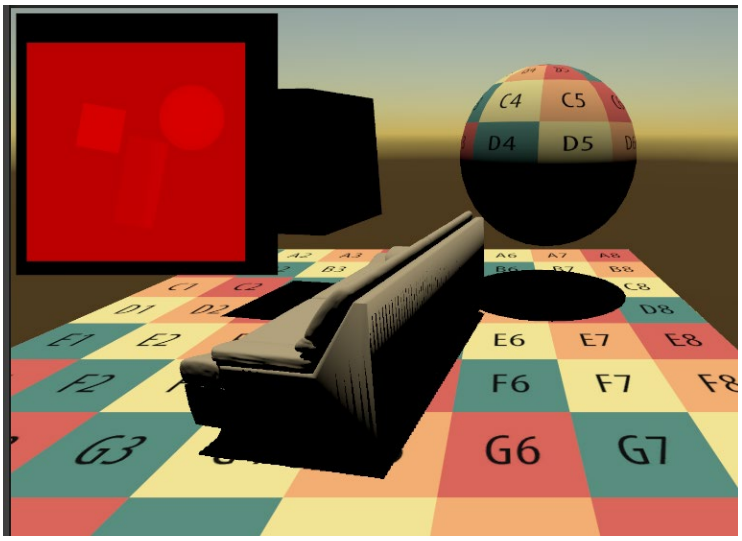
We will calculate the shadow factor. The first step is to ensure that the **shadowCoord** values are normalized correctly by doing a perspective divide. Next, we will bring it from **clip space** to **texture space**. We will sample the shadow map to get the shadow depth. We used 1.0 – value to get the correct depth value. We check if the pixel should be in shadow or in the light. (0.0 – shadow, 1.0 – light). We saturate it to clamp it between 0 and 1.

```
float ShadowCalculation(float4 fragPosLightSpace)
{
    // Transform shadow coordinates
    float3 shadowCoord = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // Transform from clip space to texture space
    shadowCoord = shadowCoord * 0.5 + 0.5;

    // Sample shadow map
    float shadowDepth = 1.0 - tex2D(_shadowMap, shadowCoord.xy).r;
    float shadowFactor = (shadowCoord.z - _shadowBias > shadowDepth) ? 1.0 : 0.0;
    // Flip the shadowFactor for proper shadowing
    shadowFactor = saturate(1.0 - shadowFactor);

    return shadowFactor;
}
```

In the fragment shader, we use the **shadowFactor** by combining it with the final color.

```
float3 finalColor = (diffuse + specularColor)
                    * _lightIntensity
                    * attenuation
                    * shadowFactor;
```

At the end of the practical, you should have something like this.

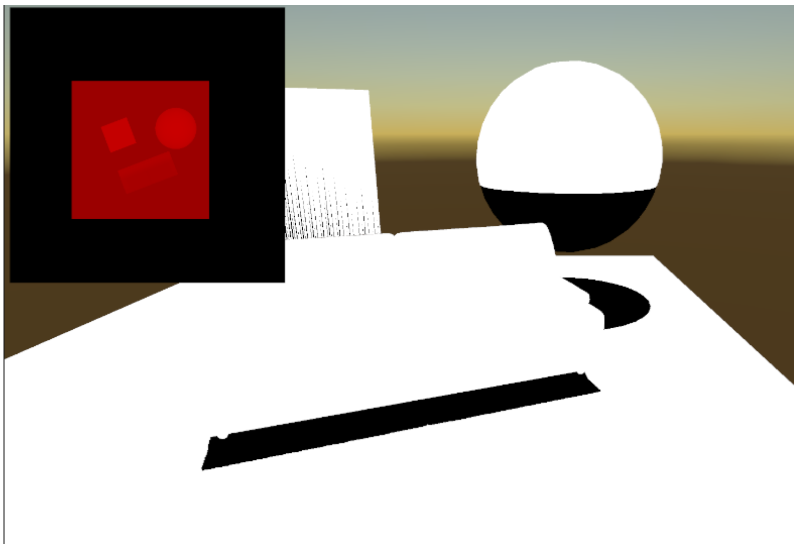


Debugging Shadow Map Checklist

Some of you might face some issue till this point where the debug plane is white. Here is a checklist to try.

- ☐ Make sure the Light Object has the Shadow Map Renderer and the Light Camera is created.
- ☐ Check the light camera. You might have used the same size as me but you might need such a huge size as your scene is smaller. Change the **size** accordingly. You check what the camera is looking at by selecting the light with the camera. Make sure that the scene is within it and it is not too small. Next check your **near** and **far** value. Also change the size of the camera.
- ☐ Check if the Shadow Factor is correct

```
return float4(shadowFactor, shadowFactor, shadowFactor, 1.0);
```



- ☐ Check if the shadowCoord is correct

```
return float4(v2f.shadowCoord.x, v2f.shadowCoord.y, v2f.shadowCoord.z, 1.0);
```