**IGB381 – Game Engine Technology**

**Workshop 8 (Week 9) – Refraction, Hard Shadows and Terrain Research**

**Aim:** To experiment with various global illumination effects for the purpose of implementation into Assignment 2.

**Objectives:**

* Create a basic refraction shader using nVidia SDK principles and explore options for possible refinement and expansion
* Develop a simple, hard shadows shader without defaulting to a pre-existing shader (i.e. UsePass)
* Begin research and experimentation for generating terrain for the purpose of addressing criteria for Assignment 2

**Preparation:**

Reviewed the Lectures regarding displacement mapping, refractions, shadows and global illumination principles and be ready to implement and explore discussed aspects. If you have started on Assignment 2, bring in your working project files and be prepared to expand on it. Additionally, download the Refraction and Hard Shadows project from Blackboard alongside this week’s workshop.

**Step 1: Refraction**

To start, open the scene Refraction in the Refraction and Hard Shadows project you downloaded from Blackboard. You should be greeted with a scene in which a simple tea pot is placed in a premade sky box.



Today we will make this teapot both reflect elements within this skybox as well as have an element of translucency and refraction. We should end up with a scene that looks similar to what was demonstrated in the Lecture materials.

Open the Refraction.shader file and scroll down to the pixel shader EnvMapPS. We will do most of our work inside this shader function. To start, let’s put in our cube map reflection, similar to how we have done in the past. Add the following to EnMapPS, including replacing the return statement with a new temporary one:

// normalize input vectors

a\_Input.norm = normalize(a\_Input.norm);

a\_Input.toVert = normalize(a\_Input.toVert);

// calculate reflection vector to use as input to cubemap

float3 reflectCol = texCUBE(\_CubeMap, reflect(a\_Input.toVert, a\_Input.norm));

return float4((reflectCol), 1.0f);

Your teapot should now look like the following, which should seem somewhat familiar at this point.



Now, we shall set up a refraction colour. This process will use a predefined refraction function and our normalized input vectors to determine the colours associated with the pixel locations of our observed object. Remove the temporary return statement and replace it with the following:

float3 refractCol;

refractCol.r = texCUBE(\_CubeMap, refract(a\_Input.toVert, a\_Input.norm, 1.1f)).r;

refractCol.g = texCUBE(\_CubeMap, refract(a\_Input.toVert, a\_Input.norm, 1.08f)).g;

refractCol.b = texCUBE(\_CubeMap, refract(a\_Input.toVert, a\_Input.norm, 1.06f)).b;

The specific values used here for refraction are retrieved from the official nVidia SDK, and cohere to the coefficients that said colour would refract through a material like glass (i.e. not all visible light travels at the exact same speed). This will give an appropriate distortion effect when utilized correctly, but feel free to return to here to adjust the values slightly.

Now we need to take into account the orientation of the surface we are observing, as well as several properties, to effectively determine a Fresnel coefficient to combine with our refraction colour. We will do this by setting up a Fresnel approximation function, also sourced from the nVidia SDK. Place this function above and separate from the EnvMapPS function:

// fresnel approximation

float fast\_fresnel(float3 I, float3 N, float3 fresnelValues) {

float power = fresnelValues.x;

float scale = fresnelValues.y;

float bias = fresnelValues.z;

return bias + pow(1.0 - dot(I, N), power) \* scale;

}

Recall that this is a function similar to what we explored in the Lecture material regarding light refraction across real surfaces. To utilize this, we need to create a separate Fresnel term to add to our final colour. Place the following back within EnvMapPS, just after where you declared your refractCol values:

float3 fresnelTerm = fast\_fresnel(-a\_Input.toVert, a\_Input.norm, float3(2.0f, 2.0f, 0.1f));

The parameters used for the final float3 parameter will also affect the Fresnel refraction strength, so feel free to manipulate them for slightly different distortion effects. To see what the result currently looks like, replace the return statement with the following:

return float4((reflectCol + refractCol + fresnelTerm) / 3.0f, 1.0f);

This final return statement now combines all elements of our reflection, refraction and Fresnel distortion model and should result in something like the following:



Notice now how the object has a translucent appearance, with shades of red/green/blue lining refracted and distorted imagery in the background. However, this may be a little hard to see, and sometimes your object does not have as high a reflective quality as its refractive one (e.g. polished glass). If we wanted to turn off the reflective properties of our shader, we could do so by introducing a Boolean-like property, and returning a different statement in our final pixel shader. To do this, add the following property to the start of your script, in the Property block:

[MaterialToggle(REFLECTION)] \_Reflection("Reflection", Float) = 0

Follow this up with a shader\_feature #pragma statement in our Pass, declaring our toggle as an element to be targeted and configured by the compiler.

#pragma shader\_feature REFLECTION

This setup is useful for turning certain shader features on and off within your games, directly via the material. To utilize this Boolean toggle, replace the final return statement in EnvMapPS once more with the following conditional logic block:

#ifdef REFLECTION

return float4((reflectCol + refractCol + fresnelTerm) / 3.0f, 1.0f);

#else

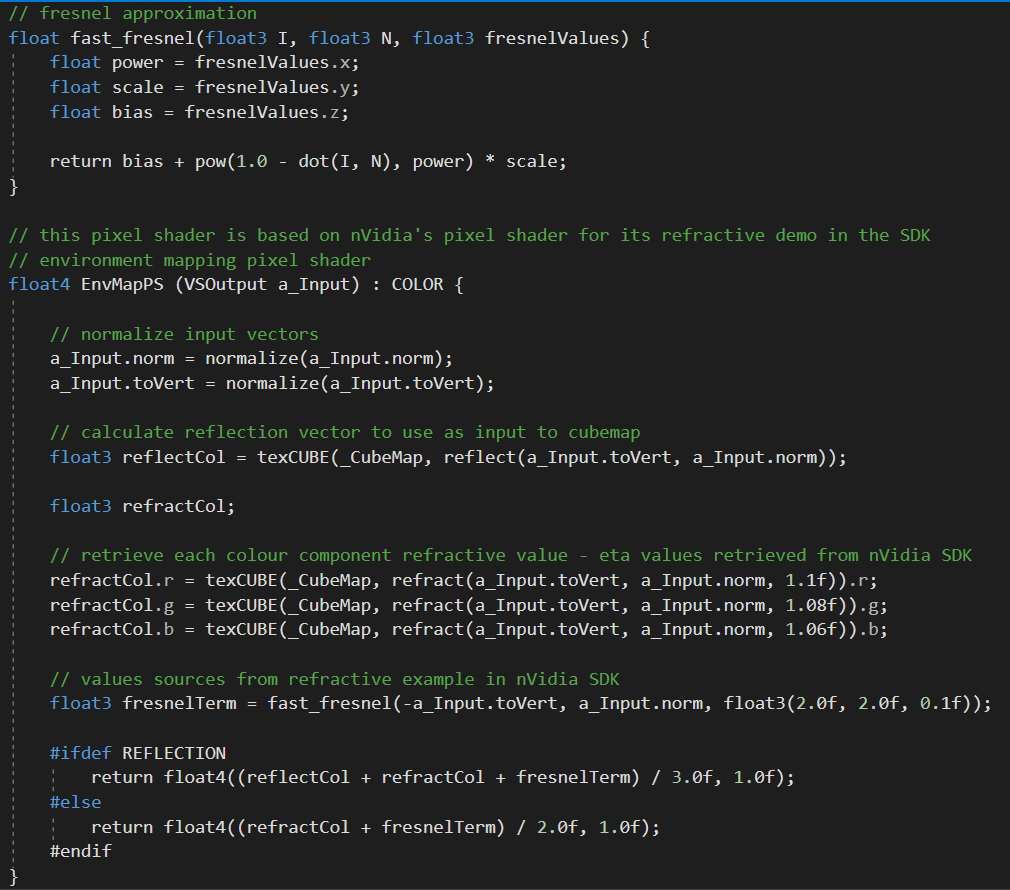
return float4((refractCol + fresnelTerm) / 2.0f, 1.0f);

#endif

Toggling the material option in the Inspector view within Unity should now also result in the following effect, removing reflective elements on the teapot:



For your reference, your script additions should now look like the following:



Ideally we would take into account the perspective our camera has upon the surface of our object (e.g. water) and interpolate the use of our reflection/Fresnel values based on said angle. The further you look out on this water, the less the object will refract and the more it would reflect. You should experiment with this by setting up a plane and determining the camera angle with some simple vector math within the vertex shader, later used in the pixel shader for manipulating our refraction and reflection values.

As an additional exercise, **c**onsider the following: Our current implementation only considers the geometry of the teapot that is facing the camera. In reality, a translucent teapot would also have an inner space, and therefore a secondary layer of ‘glass’ (or whatever it is made from) from which to refract through. How would you go about implementing said additional layer of refraction on geometry that does not by default support it? Try to implement your processes and show them to your workshop tutor.

**Step 2: Hard Shadows**

Shadows are often considered the crown jewel of shader technology. Simply put, they have the capacity to inject the most realism into a scene compared to most other shader types, primarily because shadows are everywhere in real life! Today we will (finally) get to implement a simple hard shadow casting shader.

Open the Hard Shadows scene in the current project. You should be greeted by a straightforward scene containing a directional light, a plane and a spaceship.



The shader HardShadows.shader is attached to both the spaceship and the plane. This shader is, by default, simply a diffuse shader. We shall add to this shader to include the elements needed to cast shadows. Primarily, this will consist of gathering information for rendering a ‘shadow casting’ pass, which we will also write.

To start, let’s add a #pragma and #include to our Pass to establish some links to useful helper functions and macros for storing shadow data between passes. Add this just underneath the last #include statement within the pass:

// shadow helper functions and macros

#pragma multi\_compile\_fwdbase

#include "AutoLight.cginc"

Follow this up by adding the following to the struct PS\_Input:

fixed3 ambient : COLOR1;

SHADOW\_COORDS(1) // put shadows data into TEXCOORD1

This will set up a texture register to store our shadow data, as well as provide an additional colour to calculate some ambient lighting to differentiate it from. Let’s set up this ambient lighting and save some of our shadow data for use in our pixel shader and following passes. Add this just before the return statement in the vertex shader VS\_Input:

o.ambient = ShadeSH9(half4(worldNormal,1));

// compute shadows data

TRANSFER\_SHADOW(o)

To account for the attenuation of our shadow, in our pixel shader PS\_HardShadows, add the following between the first line fixed4 col = tex2D(\_MainTex, i.uv); and the return statement:

// compute shadow attenuation (1.0 = fully lit, 0.0 = fully shadowed)

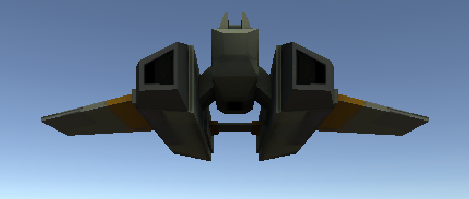
fixed shadow = SHADOW\_ATTENUATION(i);

// darken light's illumination with shadow, keep ambient intact

fixed3 lighting = i.diff \* shadow + i.ambient;

col.rgb \*= lighting;

This will calculate how light/dark our shadow is as well as account for shadows being drawn onto our object’s surfaces. You will notice that the ship is now also darkened/lightened based on the ambient lighting and its positioning, like so:



To cast shadows, we will need to write a separate pass using the data we have gathered through TRANSFER\_SHADOW(). Typically in Unity you would refer to another shader file that already has a complex shadow casting pass set up. For this workshop, we will write our own using a few helper macros. It will not account for every element of casting shadows, but it will serve the purpose of casting hard shadows from a single directional light.

Make a new Pass{} block underneath the current one, filling it with the following preliminary data:

// Shadow caster rendering pass, implemented manually using macros from UnityCG.cginc

Pass {

Tags {"LightMode"="ShadowCaster"}

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_shadowcaster

#include "UnityCG.cginc"

ENDCG

}

Our structs, vertex and pixel shaders within this pass are designed simply to extract the shadow data and apply it to surfaces in our scene on a per pixel basis. This can be done relatively easily in Shaderlab using predefined helper functions. Add the following straightforward shader functions to our final pass:

struct v2f {

V2F\_SHADOW\_CASTER;

};

v2f vert(appdata\_base v) {

v2f o;

TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET(o)

return o;

}

float4 frag(v2f i) : SV\_Target {

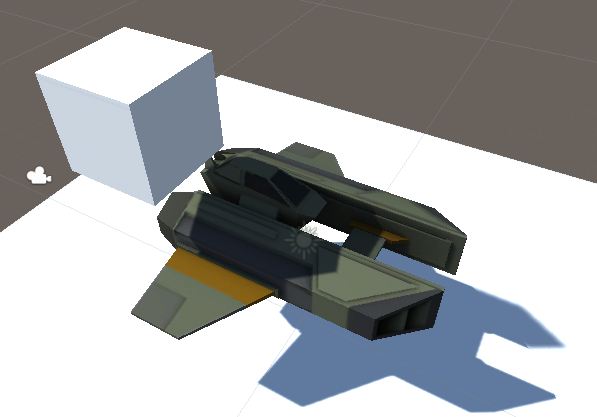
SHADOW\_CASTER\_FRAGMENT(i)

}

Returning to Unity, you should achieve the following hard shadows being drawn onto the plane below:



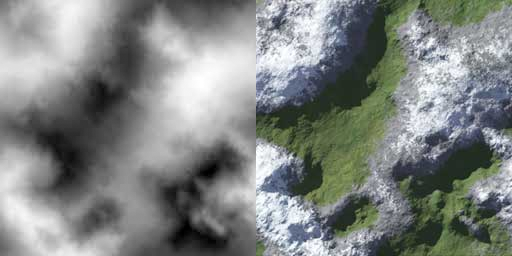
Take note that the shadow casting only works on objects that have a shadow casting pass within their shader. Creating a new cube and either applying a material using our shader or using the standard Unity shader will achieve the following results:



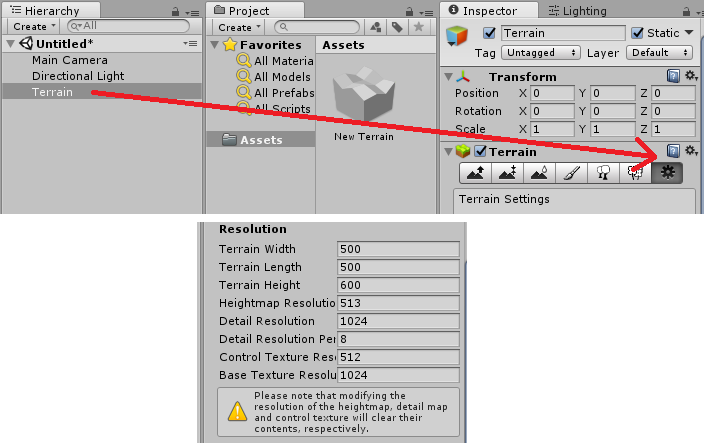
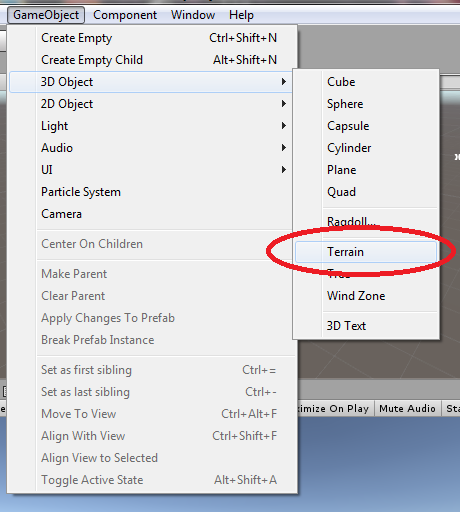
In a future workshop we will look into generating Pre-Calculated Soft Shadows (PCSS).

**Step 3: Assignment 2 - Terrain Heightmap Research**

One of the major components of Assignment 2 is creating a realistic terrain based environment without the use of Unity’s default terrain editing tools. No marks will be given for simply painting and deforming terrain using the default tools, so you will need to investigate how to create sensible terrain yourself. Ideally this should be done via the use of a few scripts that either generate or utilise a heightmap. Recall that a heightmap is like a normal map, but is more often utilized for creating large scale environments such as terrain. For example:



By now you should be somewhat familiar with deforming terrain via the use of a displacement mapping shader. However, to assist you with this process, you should initially use a basic terrain object in Unity. This can be added to your scene by going to GameObject->3D Object->Terrain, and adjusting the initial size to be something sensible within your project.



This means you initially have access to a flat Terrain game object, which itself has an assortment of adjustable properties. You may wish to look at the provided Terrain related documentation for properties which you can adjust, as well as look online for creative ways for doing so.

<https://docs.unity3d.com/ScriptReference/Terrain.html>

<https://docs.unity3d.com/ScriptReference/TerrainData.html>

This can and should be done via the use of both written C# and Shaderlab files within your Assignment. The heightmap itself can either come from a large image file you then process within Unity/Shaderlab for use, or generated via noise.

Finally, depending on your implementation, you may also need to account for the terrain not necessarily retaining its collision properties after you deform it using a heightmap. Your Assignment implementation should aim to address this, allowing the controllable avatar to traverse across the terrain, instead of clipping through it.