UNIVERSITÉ PARIS 8

2 rue de la Liberté

93 526 Saint-Denis cedex

Rapport Projet Tutoré

Jérémie Pagny N°Étudiant : 12311178

Kevin Havranek N°Étudiant : 12310320

Tuteur : Mr Farès Belhadj

2015 / 2016

Table des matières

1. Introduction

1.1 Petit résumé

Nous avions, lors du semestre précédent, découvert Unity pour pouvoir développer sur l'Oculus Rift, nous avions pour cela découvert l'outil dans son ensemble, mais aussi plus particulièrement la programmation de scripts C#.

Nous avions donc produit un petit programme, qui permettait de contrôler un hélicoptère au travers un paysage généré aléatoirement. L'utilisateur pouvait observer, grâce à l'Oculus Rift, tout autour de lui.

À la fin de cette première session, le terrain était effectivement généré aléatoirement, cependant il n'était pas encore texturé (il était tout blanc). Et pour que ceci soit effectué, nous allions utiliser les Shaders.

1.2 Unity et les Shaders

Cette deuxième session pour les projets tutorés fut donc l'occasion de découvrir comment utiliser les Shaders sous Unity. En effet si nous savions comment le faire avec la bibliothèque OpenGL (ou DirectX) directement, nous n'en avions aucune idée avec Unity (même si cet outil utilise DirectX ou OpenGL).

2. Présentation des Shaders sous Unity

2.1 Comment ça fonctionne ?

Pour commencer, au tout début lorsque nous voulions commencer à développer les shaders, nous pensions que Unity nous fournissait une interface Front-End de programmation pour les Shaders. C'est-à-dire que nous programmerions dans une syntaxe proposée par Unity, puis celui-ci se chargerait de le traduire dans un langage GLSL, HLSL ou CG.

Mais en réalité ce n'est pas aussi simple que ça, en effet si l'on souhaite programmer des Shaders, il va bien falloir développer avec un langage de Shaders. Cependant Unity encapsule ces Shaders dans une certaine syntaxe pour pouvoir faciliter l'interface entre le développement C# et les Shaders (pour par exemple définir comment envoyer une texture au Shader via Unity).

Alors même s'il existe certains types de Shaders qui permettent de faire des effets avec peu de code (les Surface Shaders), il est possible de développer des Vertex ainsi que des Fragment Shaders

2.2 Les Surface Shaders

Même si nous n'avions pas réellement utilisé ce type de Shaders, nous penssons qu'il est intéressant de les présenter, car il s'agit d'une interface permettant de générer des Vertex ainsi que des Fragment Shaders facilement.

Les Surfaces Shaders sont en fait des Shaders qui vont agir à la fois sur les Vertex ainsi que sur les Fragment. L'avantage de ces Shaders c'est qu'ils permettent de faire des calculs de couleurs sur un matériel sans pour autant avoir besoin de gérer les Vertex.

Figure 2.0 (Exemple pour texturer un objet avec les Surface Shaders)

```
Shader "Example/Diffuse Texture" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
    };
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

2.3 Les Vertex et Fragment Shaders

Malgré la présence des Surface Shaders, il reste tout à fait possible d'utiliser les Vertex et Fragment Shaders pour pouvoir agir sur le Pipeline graphique. Comme nous l'avions dit précédemment, les Shaders utilisés par Unity sont de vrais langages de Shaders. Cependant lorsque l'on souhaite en programmer avec Unity, ceux-ci sont encapsulés sur une sorte d'interface qui permettent d'avoir plusieurs informations pour Unity.

Prenons par exemple un Shader permettant d'afficher une simple couleur sur un objet (figure 2.1)

Figure 2.1 (Exemple Shader Unity simple couleur)

```
Shader "Unlit/SingleColor"
{
    Properties
    {
        // Color property for material inspector, default to white
        _Color ("Main Color", Color) = (1,1,1,1)
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            // vertex shader
            // this time instead of using "appdata" struct, just spell inputs manually,
            // and instead of returning v2f struct, also just return a single output
            // float4 clip position
            float4 vert (float4 vertex : POSITION) : SV_POSITION
            {
                return mul(UNITY_MATRIX_MVP, vertex);
            }

            // color from the material
            fixed4 _Color;

            // pixel shader, no inputs needed
            fixed4 frag () : SV_Target
            {
                return _Color; // just return it
            }
            ENDCG
        }
    }
}
```

Si l'on procède par bock, on peut d'abord observer 2 grosses sections :

– La section « Properties », qui permet ici de spécifier différentes propriétés tel que les Textures, les Couleurs, etc. qui vont être transmis depuis un script C#

– La section « SubShader » qui est la section qui va permettre de programmer tous les Shaders possibles, on peut d'ailleurs reconnaître le langage CG utilisé. Cependant viennent s'ajouter à cela quelques variables pré-processeur qui sont ici pour plusieurs fonctionnalités (spécifier la

fonction d’entrer pour les Vertex et Fragment Shaders, aider au développement, etc.).

2.4 Quel langage choisir ?

On peut donc choisir 3 langages de Shaders, GLSL, HLSL ou CG :

– Le langage CG qui est le langage de Shaders de Nvidia, semble être le préféré de Unity car celui-ci est utilisé pour les exemples dans la documentation de Unity, CG est un langage qui se rapproche énormément de HLSL au niveau de la syntaxe et au niveau du fonctionnement.

– Le langage HLSL qui est le langage de Shaders pour DirectX, seul problème de ce langage, c’est qu’il y a très peu d’exemples sur internet, comparé a son concurrent GLSL.

– Le langage GLSL qui est le langage de Shaders pour OpenGL, à l’inverse de HLSL il y a beaucoup d’exemples sur internet, cependant Unity montre très peu d’exemple avec son interface.

Même si durant nos études nous avons surtout étudié et développez en GLSL, nous avons décidé d’utiliser CG, car c’est le seul langage qui a des exemples avec Unity. Et comme il y a beaucoup de fonctionnalités pré-processeur avec Unity nous pensons que c’était le choix le plus judicieux.

3. Shader pour le terrain

3.1 Pourquoi en avoir besoin ?

Un Shader est un programme exécuté par la carte graphique et qui va rendre possible plusieurs effets graphiques tels que les explosions ou simplement de texturer des objets.

C’est dans le but de texturer notre terrain ainsi que de lui appliquer un effet de toon que nous avons besoin des Shaders.

Notre Shader va donc texturer notre terrain avec trois textures qui vont être appliquées aux parties du terrain.

3.2 Côté C#

Pour pouvoir utiliser un Shader avec Unity il faut passer par un script, qui chez nous est en C#. Dans ce script nous devons déclarer un objet de type « Shader » qui est notre Shader, nous avons également besoin de trois variables de type « Texture2D » pour stocker nos textures.

Il nous faut ensuite un objet de type « Material », c’est cet objet qui va faire le lien entre le terrain et le Shader. C’est via l’objet « Material » que nous envoyons nos textures et autres variables au Shader.

Pour l’effet de Toon Shading nous générons une texture de niveaux de gris que nous envoyons au Shader.

3.3 Côté Shader

Le Shader va recevoir les textures ainsi que les données de la lumière qui sont sa couleur, son intensité ainsi que sa direction. Nous avons dû recréer l’éclairage de Phong, car nous n’utilisions plus le Shader par défaut, cependant nous avons fait du Blinn Phong qui est une variante de Phong qui allège le calcul du reflet.

Nous calculons l’éclairage de Phong avec la couleur de la lumière, le vecteur de la normale, le vecteur de direction de la lumière, le Shininess ainsi que le vecteur de vue. Pour texturer le terrain nous regardons la position du pixel et selon celle-ci nous appliquons la texture voulue. Les textures sont mixées pour éviter un changement brutal de texture.

Pour le Toon Shading nous avons une texture de niveau de gris dans laquelle nous choisissons une valeur selon l’intensité de la lumière que reçoit le pixel.

4. Les Shaders Post-Effects

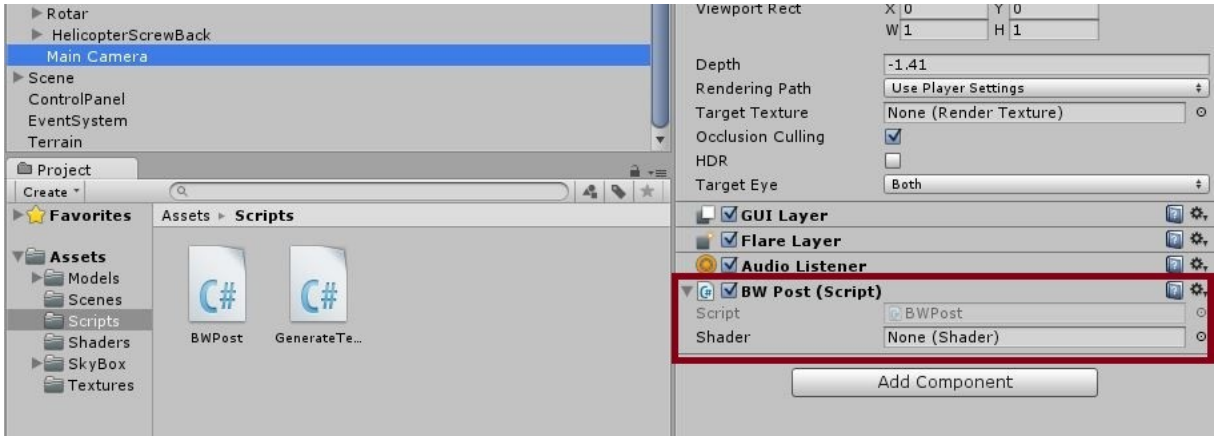
4.1 Pourquoi en avoir besoin ?

Nous n'en avions pas réellement besoin, cependant lorsque nous avions voulu commencer à faire du Toon Shading, nous avions fais une erreur de compression. En effet nous pensions au début que cet effet était appliqué sur une image et non sur scène 3D (c'est bien entendu tout le contraire). Mais cette erreur fut l'occasion de voir comment effectuer des effets sur une scène 3D en temps réel.

4.2 Côté C#

Pour pouvoir appliquer un Shader Post-Effect, il faut tout d'abord avoir une caméra de présente dans la scène. C'est en effet sur cet objet que l'on va appliquer un script C# (Figure 4.0) qui va gérer le Shader.

Figure 4.0



Ce script doit avoir une méthode « OnRenderImage », qui est une méthode qui sera appelée à chaque loop d'images. Cette dans cette méthode que le Shader sera appelé pour y appliquer son effet.

4.3 Côté Shader

Pour tous les Shaders Post-Effect, il y a certaines règles à respecter, si l'on prend par exemple le Shader suivant (Figure 4.0) :

Figure 4.0 (Exemple d'un Shader Post-effect, permettant d'avoir l'affichage de la scène en noir et blanc)

```
Shader "Hidden/BW"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        // No culling or depth
        Cull Off ZWrite Off ZTest Always

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
            };

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
                o.uv = v.uv;
                return o;
            }

            sampler2D _MainTex;

            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 col = tex2D(_MainTex, i.uv);
                col = col.r + col.g + col.b / 3;
                return col;
            }
            ENDCG
        }
    }
}
```

Il faut tout d'abord une propriété (« _MainTex » dans ce cas-là) qui sera ni plus ni moins notre texture en entrée, ou plus simplement notre scène 3D pour laquelle nous souhaitons y appliquer un effet. Il faut ensuite une structure (nommé ici « appdata »), qui va permettre de formater les données que le Vertex Shader va recevoir en entrée, il faut que cette structure puisse accueillir les vertex (pour pouvoir dessiner le rectangle qui fera office d'écran), ainsi que des coordonnées de textures (pour pouvoir appliquer la texture sur le rectangle).

Le vertex sera donc là pour simplement afficher la nouvelle image, car il ne s’agit pas là d’avoir à manipuler des Meshs. Une fois que ceci est fait, le plus important est donc le Fragment Shader qui est l’endroit où l’on peut commencer à y appliquer des effets.

5. Conclusion

5.1 Ce qui a été fait

Lors de ce second semestre nous avons donc découvert les Shaders sous Unity dans le langage CG. Nous avons réussi à texturer notre terrain avec plusieurs textures selon la hauteur de celui-ci. Nous avons également recréé la gestion de la lumière avec un Blinn Phong ainsi qu’un effet de Toon Shading.

5.2 Ce qui n’a pas été fait

Nous voulions améliorer la génération de terrain pour qu’elle soit plus réaliste, cependant cette amélioration n’a pas été faite.