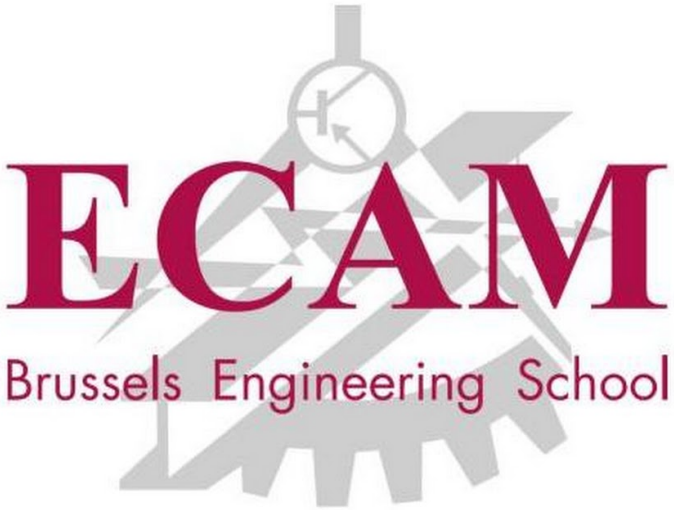

Synthèse d'images

OpenGL - Unity

Auteurs :
Julien Kessels
Saïkou Barry

Enseignant :
Quentin Lurkin

1er juin 2018

1 Objectifs

L'objectif du projet est de mettre en place une simulation de fluide basée sur la méthode FLIP/PIC et le code de base 3D-Engine-Base.

2 Problèmes rencontrés

Etant des utilisateurs de mac, nous étions limités par les capacités de ces machines. En effet, le code de base mis à disposition par Monsieur Lurkin était non fonctionnelle sur nos machines car ce code utilisait des compute Shaders qui sont disponible sur OpenGL 4.3 et plus. Or, Apple a limité OpenGL à la version 4.1. Pour faire fonctionner le code de base, il aurait donc fallu que nous récrivions le code du compute Shader sous forme de surface et vertex Shader ce qui demande un temps énorme et pas nécessairement le but du projet.

Il nous a donc été demandé d'explorer d'autres solutions possible et d'effectuer un projet d'exemple par rapport à cette solution. Nous avons donc opté pour le moteur de jeu multi-plateforme Unity.
Dans la suite du rapport, nous allons développer, en mesure du possible sur Unity et mac, un Shader pour simuler la neige sur un rocher.

3 Le projet sur Unity

3.1 Le rendu sur Unity

Le rendu dans Unity utilise des **matériaux**, des **Shaders** et des **Textures**. Tous les trois ont une relation étroite.

Les *matériaux* définissent la manière dont une surface doit être rendue, en incluant des références aux Textures utilisées, des informations de mosaïque, des couleurs et plus encore. Les options disponibles pour un matériau dépendent du Shader utilisé par le matériau.

Les *Shaders* sont de petits scripts qui contiennent les calculs mathématiques et les algorithmes pour calculer la couleur de chaque pixel rendu, en fonction de l'entrée d'éclairage et de la configuration matérielle.

Les *Textures* sont des images bitmap. Un matériau peut contenir des références à des Textures, de sorte que le matériau du matériau puisse utiliser les Textures lors du calcul de la couleur de surface d'un objet GameObject. En plus de la couleur de base (Albedo) de la surface d'un GameObject, les Textures peuvent représenter de nombreux autres aspects de la surface d'un matériau, tels que sa réflectivité ou sa rugosité.

Un matériau spécifie un Shader spécifique à utiliser et le Shader utilisé détermine les options disponibles dans le matériau. Un Shader spécifie une ou plusieurs variables de Texture qu'il s'attend à utiliser, et l'inspecteur de matériel dans Unity nous permet d'affecter nos propres Textures à ces variables de Texture.

3.2 Shader sur Unity

Un *Shader* dans Unity peut être écrit de l'une des trois façons suivantes :

— en tant que Shaders de surface (**surface Shader**)
C'est la meilleure option si notre Shader doit être affecté par les lumières et les ombres. Les Shaders de surface facilitent l'écriture de Shaders complexes de manière compacte - c'est un niveau d'abstraction supérieur pour l'interaction avec le pipeline d'éclairage d'Unity. La plupart des Shaders de surface prennent automatiquement en charge l'éclairage avant et différé. On écrit les Shaders de surface en **Gg/HLSL**, et Unity gère le reste pour générer automatiquement le code nécessaire à partir de cela.

N'utilisez pas de Surface Shaders si votre Shader ne fait rien avec des lumières.

— comme Shaders de sommet et de fragment (**fragment and vertex Shader**)
On utilise les Shaders de sommet et fragment si nous n'avons pas besoin d'interagir avec l'éclairage, ou si nous avons besoin d'effets plus complexe que les Shaders de surface ne peuvent pas gérer. Ces Shaders sont également écrits en **Gg/HLSL**.

— comme Shaders à fonction fixe. (**function Shader**)
Les Shaders de fonction fixes sont des Shaders utilisé pour des effets très simples. Il est conseillé d'écrire des Shaders programmables, car cela permet beaucoup plus de flexibilité. Les Shaders de fonctions fixes sont entièrement écrits dans un langage appelé **ShaderLab**, qui est similaire aux fichiers *.FX* de Microsoft ou au *CgFX* de NVIDIA. En interne, tous les Shaders à fonction fixe sont convertis en Shaders de fragment et sommet au moment de l'importation des Shaders.

Quel que soit le type que nous choisissons, le code du Shader sera toujours enveloppée dans un langage appelé **ShaderLab**, qui est utilisé pour organiser la structure de Shader. Ça ressemble à ça :

ShaderLab syntax

Comme dit plus haut, tous les fichiers Shaders dans Unity sont écrits dans un langage déclaratif appelé "ShaderLab". Dans le fichier, une syntaxe nested-braces déclare diverses

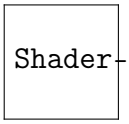

FIGURE 1 – Structure d'un Shader - 01

choses qui décrivent le Shader - par exemple, quelles propriétés de Shader doivent être affichées dans l'inspecteur de matériaux ; quel genre de mode de fusion utiliser, etc . ; et le "Shader code" actuel est écrit dans des fragments **CGPROGRAM** dans le même fichier Shader.

Shader est la commande racine d'un fichier Shader. Chaque fichier doit définir un (et un seul) Shader. Il spécifie comment les objets dont le matériau utilise ce Shader sont rendus.

```
Shader "name" { [Properties] SubShaders [Fallback] [CustomEditor] }
```

Définit un Shader. Il apparaîtra dans l'inspecteur de matériaux listé sous le nom donné (ici pour l'exemple, *name*). Les Shaders peuvent éventuellement définir une liste de propriétés qui apparaissent dans l'inspecteur de matériaux. Après cela vient une liste de SubShaders, et éventuellement un fallback et/ou une déclaration d'éditeur personnalisé.

— **Propriétés**
Les Shaders peuvent avoir une liste de propriétés. Toutes les propriétés déclarées dans un Shader sont affichées dans l'inspecteur de matériaux dans Unity. Les propriétés typiques sont la couleur de l'objet, les Textures ou simplement les valeurs arbitraires à utiliser par le Shader. Les propriétés sont partagées entre tous les sub-Shaders.

— **SubShaders & Fallback**
Chaque Shader est composé d'une liste de sous-Shaders. Nous devons en avoir au moins un. Lors du chargement d'un Shader, Unity parcourt la liste des sous-Shaders et sélectionne le premier supporté par la machine de l'utilisateur final. Si aucun sous-ensemble n'est pris en charge, Unity essaiera d'utiliser le Shader de secours (Fallback).
Les différentes cartes graphiques ont des capacités différentes. Cela soulève un problème éternel pour les développeurs de jeux ; Nous voulons supportez le plus possible de machines. C'est là que les sous-Shaders entrent en jeu. Nous créons un sous-ensemble qui a tous les effets graphiques fantaisistes dont nous rêvons, puis nous pouvons ajouter plus de sous-Shaders pour les anciennes cartes. Ces sous-Shaders peuvent implémenter l'effet que nous voulons de manière plus lente, ou ils peuvent choisir de ne pas implémenter certains détails.

3.3 Notre Shader - Snow

Nous avons implémenter un simple Shader appelé Snow. Voici ce que le Shader fait :
— Au fur et à mesure que le niveau de neige augmente, nous transformons les pixels qui font face à la direction de la neige en une couleur de neige plutôt que la Texture du matériau.
— Lorsque le niveau de neige augmente, nous déformons le modèle pour qu'il soit plus grand, surtout du côté de la neige.

Properties

Nous avons plusieurs propriétés pour notre Shader.
— une variable **Snow** qui sera la quantité de neige qui recouvre la roche, elle est toujours dans la plage 0..1
— une variable **SnowColor** pour notre neige (évitez le jaune) qui par défaut est blanc
— une variable **SnowDirection** à partir de laquelle la neige tombe (par défaut, elle tombe vers le bas, donc notre vecteur d'accumulation est droit)
— une variable **SnowDepth** pour notre neige que nous utiliserons lorsque nous modifions les sommets qui est dans la plage 0..0.3
— une variable **Wetness** pour indiquer combien nous devrions mélanger la Texture neige avec la Texture de base du modèle
— une variable **Bump** qui sert à donner du relief à la Texture
— une variable **MainTex** qui stock la Texture du modèle

SubShader

Dans cette section, nous allons décortiquer le fonctionnement du "code" qui nous intéresse vraiment.

Tags
Les Shader de surface peut être décoré avec un ou plusieurs tag. Ces balises définissent les éléments qui permettent au hardware de décider quand appeler le Shader.

Dans notre cas, nous avons :

```
Tags {"RenderType" = "Opaque"}
```

qui demande au système de nous appeler quand il fait le rendu de la géométrie Opaque.

Variables
Si nous voulons utliser les propriétés declarées plus haut dans le Shader, nous devons créer

```
Shader "Custom/SnowShader" {
    Properties{
        _MainTex("Base (RGB)", 2D) = "white" {}
        _Bump("Bump", 2D) = "bump" {}
        _Snow("Snow Level", Range(0,1)) = 0
        _SnowColor("Snow Color", Color) = (1.0,1.0,1.0,1.0)
        _SnowDirection("Snow Direction", Vector) = (0,1,0)
        _SnowDepth("Snow Depth", Range(0,0.2)) = 0.1
        _Wetness("Wetness", Range(0, 0.5)) = 0.3
    }
    SubShader{
        Tags{ "RenderType" = "Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert vertex:vert

        sampler2D _MainTex;
        sampler2D _Bump;
        float _Snow;
        float4 _SnowColor;
        float4 _SnowDirection;
        float _SnowDepth;
        float _Wetness;

        struct Input {
            float2 uv_MainTex;
            float2 uv_Bump;
            float3 worldNormal;
            INTERNAL_DATA
        };

        void vert(inout appdata_full v) {
            //Convert the normal to world coortinates
            float4 sn = mul(UNITY_MATRIX_IT_MV, _SnowDirection);

            if (dot(v.normal, sn.xyz) >= lerp(1,-1, (_Snow * 2) / 3)) {
                v.vertex.xyz += (sn.xyz + v.normal) * _SnowDepth * _Snow;
            }
        }

        void surf(Input IN, inout SurfaceOutput o) {
            half4 c = tex2D(_MainTex, IN.uv_MainTex);
            o.Normal = UnpackNormal(tex2D(_Bump, IN.uv_Bump));
            half difference = dot(WorldNormalVector(IN, o.Normal), _SnowDirection.xyz) - lerp(1,-1,_Snow);;
            difference = saturate(difference / _Wetness);
            o.Albedo = difference * _SnowColor.rgb + (1 - difference) *c;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

FIGURE 2 – Snow Shader code

des variables avec exactement le même nom dans le subShader.

Structure
Le Shader de surface est placé dans le bloc **CGPROGRAM..ENDCG**, comme n'importe quel autre Shader. Il utilise la directive pragma surface ... pour indiquer qu'il s'agit d'un Shader de surface.

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

— **surfaceFunction** - la fonction qui a le code du Shader de surface. La fonction doit avoir la forme d'un void surf (Input IN, inout SurfaceOutput o), où Input est une structure que nous avons définie.
— **lightModel** - modèle d'éclairage à utiliser

Dans notre cas, nous avons un paramètre optionnel en plus **vertex :vert** qui est une fonction de modification de vertex personnalisée. Cette fonction est appelée au début du vertex Shader généré et peut modifier ou calculer des données.

Nous avons donc défini une *fonction de surface* qui prend en entrée les UV et les données dont nous avons besoin dans un objet **Input**, et remplit la structure de sortie **SurfaceOutput**. SurfaceOutput décrit essentiellement les propriétés de la surface (couleur albédo, normale, émission, spécularité, etc).

Le compilateur Surface Shader détermine ensuite quelles entrées sont nécessaires, quelles sorties sont remplies et ainsi de suite, et génère des Shaders de vertex et de pixel réels, ainsi que des passes de rendu pour gérer le rendu en avant et en différé.

4 Conclusion

Ce rapport a introduit doucement les deux types de Shaders dans Unity et explique comment les utiliser l'un et l'autre et comment les mettre en place en détail. Nous avons réussi à écrire un simple Shader pour simuler la neige sur un rocher. Nous nous sommes également demandés s'il était possible d'utiliser les compute Shaders sur Unity et de les faire fonctionner sur mac.

Les essais réalisés sur Unity qui impliquaient des compute Shaders n'étaient pas non plus concluants. De par nos recherches, il serrait quand même possible de faire tourner des compute Shaders via Unity sous la condition de respecter l'API Metal, framework pour la modélisation 3D développé par Apple. De tels programmes sont malheureusement très rares et jusqu'à ce jour, nous n'en avons pas trouvé.