

# Compte Rendu TP4 - Textures

COMBOT Evan

07/12/2023

## Programmation 3D

Université de Montpellier - Master 1 IMAGINE

### Résumé

Ce TP à pour but de manier les textures et créer des effets à l'aide de celles ci comme faire du bump mapping, du normal mapping, de créer une skybox ou encore de faire du PBR.

## Préambule

Pour que le programme fonctionne correctement sur votre ordinateur, pensez à supprimer les donnés du dossier build puis à rebuild (sh build.sh) le projet.

## Exercice 1 : Plaillage d'une texture

Pour plaquer une texture à un plan, j'ai importé le modèle 3D d'un plan, j'ai ensuite chargé la texture dans la fonction init() comme ceci :

```
1 m_texture = loadTexture2DFromFilePath("data/Textures/rock_wall_10_diff_1k.jpg");
```

Puis j'ai "bind" la texture dans la fonction internalBind() comme ceci :

```
1 if (m_texture != -1) {
2     glBindTexture(GL_TEXTURE_2D, m_texture1);
3     glActiveTexture(GL_TEXTURE0);
4     glUniform1i(getUniform("colorTexture1"), GL_TEXTURE0);
5 }
```

Dans le fragment shader, j'ai dé-commenté cette ligne pour pouvoir afficher la texture :

```
1 FragColor = texture(colorTexture1, o_uv0) * color;
```



FIGURE 1 – Plaillage d'une texture sur un plan

## Exercice 2 : Normal Mapping

Pour faire du normal mapping, j'ai d'abord créé un éclairage basé sur le modèle de Phong puis pour pouvoir utiliser la carte de normales, je calcule dans le vertex shader la matrice TBN (Tangente, Bitangente, Normale) de taille 3x3, je calcule ensuite la transposée de cette matrice

puis je calcule la position de la lumière, de la vue et de la position du modèle dans l'espace tangent. Dans le fragment shader, j'utilise la carte de normales que j'ai normalisé à la place de la normale du modèle de Phong tout en ayant remplacé les éléments nécessaires par les éléments de l'espace tangent. Shader : vertex/fragment\_NormalMapping.glsl

Vertex.glsl :

```

1 #version 330 core
2
3 layout(location = 0) in vec3 position;
4 layout(location = 1) in vec3 normal;
5 layout(location = 2) in vec3 tangent;
6 layout(location = 3) in vec2 uv0;
7
8 uniform mat4 model;
9 uniform mat4 view;
10 uniform mat4 projection;
11 uniform vec3 lightPos;
12 uniform vec3 viewPos;
13
14
15 out vec3 o_positionWorld;
16 out vec3 o_normalWorld;
17 out vec2 o_uv0;
18 out vec3 TangentLightPos;
19 out vec3 TangentViewPos;
20 out vec3 TangentFragPos;
21
22
23 void main() {
24     mat3 normalMatrix = mat3(transpose(inverse(model)));
25     o_uv0 = uv0;
26     vec4 positionWorld = model * vec4(position, 1.0);
27     o_positionWorld = positionWorld.xyz;
28     o_normalWorld = normalMatrix * normal;
29
30     vec3 Pos = vec3(model * vec4(position, 1.0));
31
32     vec3 T = normalize(vec3(model * vec4(tangent, 0.0)));
33     vec3 N = normalize(vec3(model * vec4(normal, 0.0)));
34     T = normalize(T - dot(T, N) * N);
35     vec3 B = cross(N, T);
36
37     mat3 TBN = transpose(mat3(T, B, N));
38     TangentLightPos = TBN * lightPos;
39     TangentViewPos = TBN * viewPos;
40     TangentFragPos = TBN * Pos;
41
42     gl_Position = projection * view * positionWorld;
43 }
```

Fragment.glsl :

```

1 #version 330 core
2 #define GLSLIFY 1
3 #define GLSLIFY 1
4
5 in vec3 o_positionWorld;
6 in vec3 o_normalWorld;
7 in vec2 o_uv0;
8 out vec4 FragColor;
9
10 uniform sampler2D colorTexture1;
11 uniform sampler2D colorTexture2;
12
13 in vec3 TangentLightPos;
14 in vec3 TangentViewPos;
15 in vec3 TangentFragPos;
16
17 void main() {
18     vec3 lightColor = vec3(1.);
19
20     float NormalStrength = 1.0f;
21
22     vec3 normal = texture(colorTexture2, o_uv0).rgb;
```

```

23     normal = normalize(normal * 2.0 - 1.0);
24     normal.x *= NormalStrength;
25     normal.y *= NormalStrength;
26
27     // ambient
28     float ambientStrength = 1.0f;
29     vec3 ambient = ambientStrength * lightColor;
30
31     // diffuse
32     float diffuseStrength = 1.0f;
33     //vec3 norm = normalize(normal);
34     vec3 lightDir = normalize(TangentLightPos - TangentFragPos);
35     float diff = max(dot(normal, lightDir), 0.0);
36     vec3 diffuse = diffuseStrength * diff * lightColor;
37
38     // specular
39     float specularStrength = 1.0f;
40     vec3 viewDir = normalize(TangentViewPos - TangentFragPos);
41     vec3 reflectDir = reflect(-lightDir, normal);
42     float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
43     vec3 specular = specularStrength * spec * lightColor;
44
45
46     vec3 result = (ambient + diffuse + specular) * texture(colorTexture1, vec2(
47         o_uv0.x, 1 - o_uv0.y)).rgb;
48     FragColor = vec4(result, 1.0f);
49 }
```

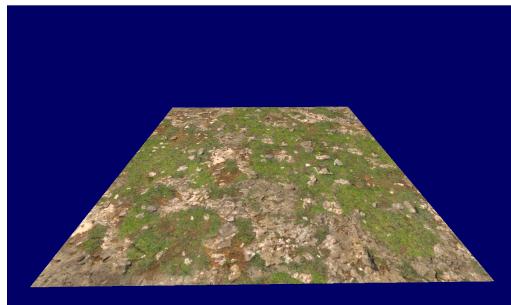


FIGURE 2 – Normal Mapping d'un plan avec normalStrength = 1.0

### Exercice 3 : Skybox

Pour créer une skybox, j'ai d'abord créé la fonction : GLuint loadCubemap(std::vector<std::string> facesPath) qui permet de charger les textures de la skybox. Ensuite, j'ai crée la fonction : initSkybox() dans le fichier Context.cpp, j'ai créé un cube, j'ai chargé le shader de la skybox puis j'ai chargé les textures à l'aide de la fonction loadCubemap(). J'envoie ensuite les données au shader. Enfin, j'ai créé la fonction : drawSkybox() qui permet d'afficher la skybox. J'appelle ensuite ces fonctions dans le main. Shaders : vertex/fragment\_Skybox.glsl (pour la skybox) / vertex/fragment\_BaseReflection.glsl (pour la réflexion) / vertex/fragment\_NormalMappingWithReflections (pour le normal mapping + réflexion)



FIGURE 3 – Skybox + plan (Normal Mapping)

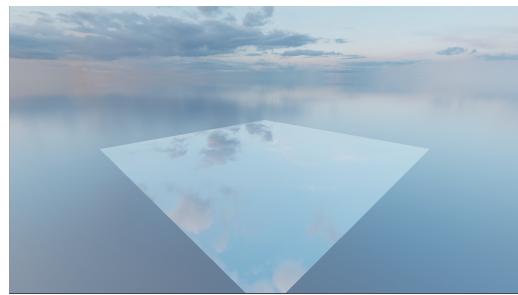


FIGURE 4 – Skybox + plan (Réflexion)



FIGURE 5 – Skybox + plan (Normal Mapping)



FIGURE 6 – Skybox + plan (Réflexion)



FIGURE 7 – Skybox + plan (Normal Mapping + Réflexion)

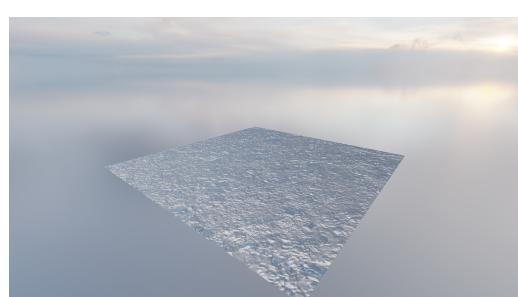


FIGURE 8 – Skybox + plan (Normal Mapping + Réflexion)

## Exercice 4/5 : PBR (basique et texturé)

J'ai créé 2 shaders pour faire du PBR, l'un sans texture et l'autre avec textures. Je n'ai pas réussi à faire de l'émissive mais j'ai réussi à ajouter l'occlusion ambiante. Shaders : vertex/fragment\_PBR\_Basic (PBR classique) / vertex/fragment\_PBR\_Textured (PBR avec textures). Le PBR basique permet de reproduire des matériaux réalistes comme de l'or ou du plastique. Le PBR avec les textures permet de reproduire de manière réaliste les matériaux en prenant en compte l'albedo, la carte de normales, la carte "roughness", la carte de l'occlusion ambiante et un paramètre qui permet de représenter le "métallique" du matériau.

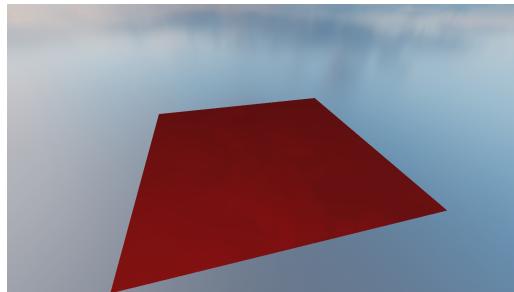


FIGURE 9 – Plan rouge un petit peu métallisé

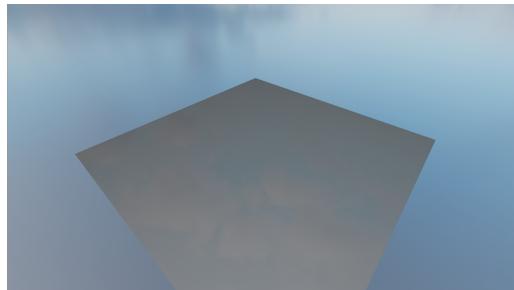


FIGURE 10 – Plan blanc un petit peu métallisé

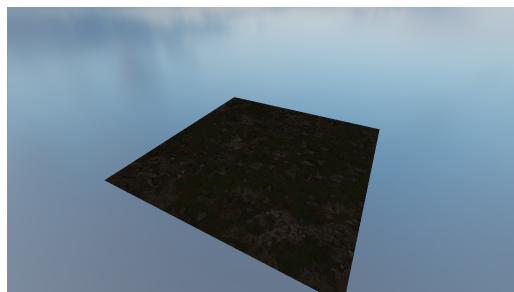


FIGURE 11 – Plan texturé (avec du normal mapping)

## Avis

J'ai trouvé ce TP très intéressant car c'est le côté réaliste de la 2D, 3D qui m'intéresse le plus (PBR, simulation, etc...).