

TP8 et 9 Programmation 3D : Textures, cubemaps et PBR

Marie BOCQUELET : 21702374

24 December 2022

I Organisation :

Pour commencer, parlons de l'organisation que j'ai mise en place pour ce tp. Au début de la prise de connaissance du TP, je ne comprenais pas la base de code et n'arrivait pas à démarrer. En effet, ce n'est que mon propre avis mais il est compliqué de s'adapter à un code dont nous ne sommes pas l'auteur. Ainsi, afin de mieux comprendre la base de code, j'ai d'abord refait une base de code moi même from scratch. Cela m'a permis de bien reposer les bases d'OpenGL, à savoir la création d'une fenêtre, la mise en place des shaders... Cette étape qui a été très bénéfique, m'a pris environ une dizaines d'heures.

Une fois ceci fait, j'ai tenté de reprendre la base de code donnée, et effectivement tout me paraissait beaucoup plus clair. J'ai alors pu passer aux choses sérieuses. Concernant l'organisation, elle était très simple : pendant plus d'une semaine, chaque jour je travaillais et tentais d'avancer au mieux. Si je voyais que je passait trop de temps sur une partie, je passais à la suivante, sauf si la précédente était indispensable pour la suite.

Une autre astuce que j'ai mise en place, est que, pour éviter de m'embrouiller en rajoutant trop de choses dans mon code, je téléchargeais une nouvelle base de code à chaque étape. Ce qui fait que j'ai trois codes, un qui concerne les textures et normal map, un autre qui concerne la skybox, et enfin un dernier pour le PBR.

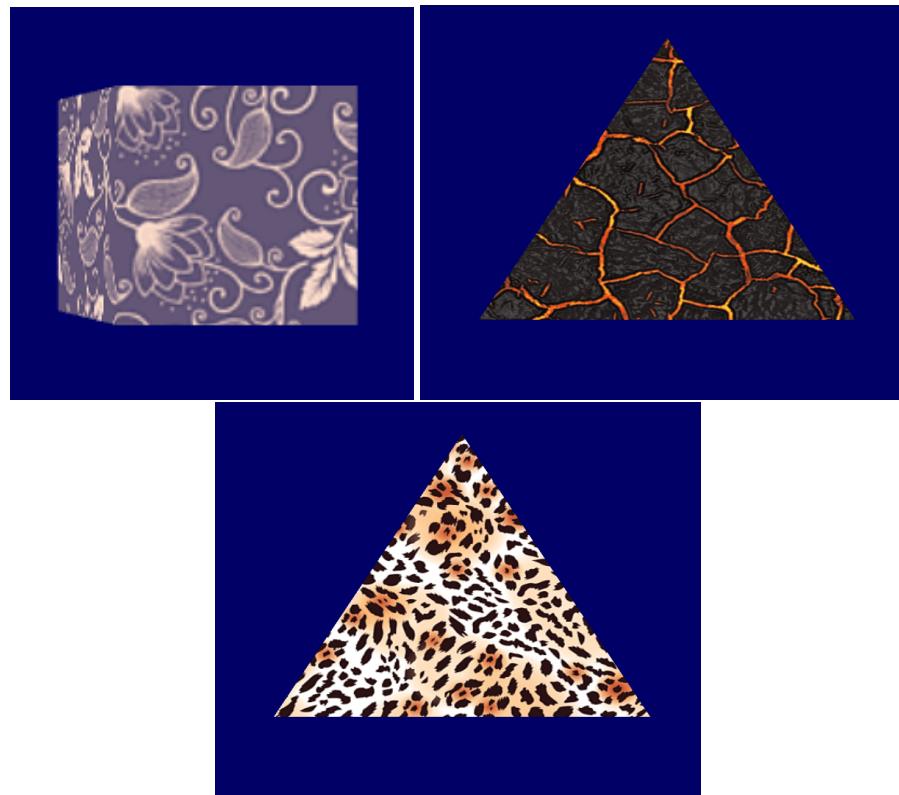
Finalement, j'ai gardé la rédaction du rapport pour le dernier jour.

II TP8 :

II.1 Niveau 0 : plaquage de texture

Dans le niveau 0 du premier TP, nous devions plaquer une texture. Pour faire ceci j'ai uniquement modifié les fichiers nommés *Material*, *Vertex* et *Fragment*. Donc pour plaquer une texture, il faut tout d'abord créer une

variable que j'ai appelée **diffuse-texture** dont on récupèrera l'image concernée grâce à la fonction *LoadTexture2DFromFilePath()*. Une fois ceci fait, il est important de penser à supprimer la texture. Maintenant, il faut compléter les fragment et vertex shaders. Nous allons avoir besoin de positions et de coordonnées de textures dans le vertex shader qui seront ensuite envoyées au fragment shader dans lequel la fonction *texture()* de GLSL sera utilisée pour donner aux coordonnées de texture, la texture appropriée. Enfin, la dernière étape consiste à faire le lien entre notre code et le fragment shader. En effet, il est nécessaire de donner au code la location de notre texture dans le shader, et bien sûr d'activer la texture, sans quoi elle ne sera jamais visible. C'est ainsi que j'ai pu obtenir les résultats suivants :



Pour ce niveau là, je n'ai pas rencontré de difficultés spécifiques, mais j'ai appris à plaquer une texture sur un objet.

II.2 Niveau 1 : les normal map

Dans cette section, une fois que l'on a une texture en 2D, on voudrait y ajouter du relief. Pour cela, nous allons utiliser ce qui s'appelle une normal map. C'est une "texture" basée uniquement sur du relief que l'on va appliquer en plus de notre texture colorée pour donner du relief à notre objet.

Pour ceci, nous avons d'abord besoin de mettre en place l'illumination de Phong, sans quoi nous ne pourrions rien voir sur notre objet. Je me suis inspirée de learnOpenGL pour arriver à effectuer ce travail. La majorité

se fait dans les vertex et fragment shader :

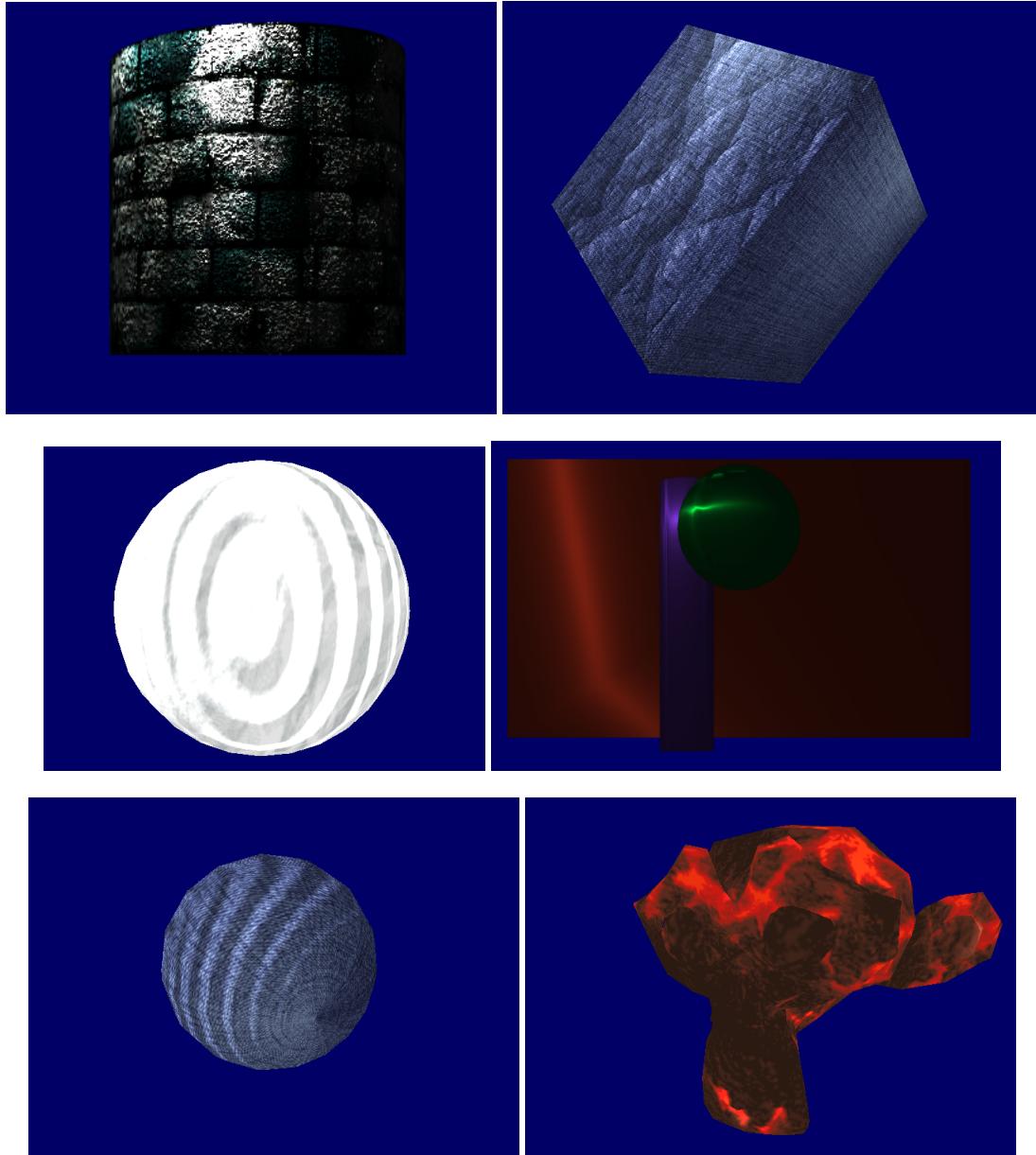
-Vertex Shader :

```
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
12 out vec3 worldSpace_fragmentPosition;
13 out vec2 o_uv0;
14 out vec3 tangentSpace_lightPosition;
15 out vec3 tangentSpace_viewPosition;
16 out vec3 tangentSpace_fragmentPosition;
17
18 uniform vec3 light_position;
19 uniform vec3 view_position;
20
21 void main() {
22     worldSpace_fragmentPosition = vec3(model*vec4(position, 1.0));
23     o_uv0 = uv0;
24
25     mat3 normalMatrix = transpose(inverse(mat3(model)));
26     vec3 T = normalize(normalMatrix*tangent);
27     vec3 N = normalize(normalMatrix*normal);
28     T = normalize(T - dot(T,N) * N);
29     vec3 B = cross(N, T);
30
31     mat3 TBN = transpose(mat3(T,B,N));
32     tangentSpace_lightPosition = TBN * light_position;
33     tangentSpace_viewPosition = TBN * view_position;
34     tangentSpace_fragmentPosition = TBN * worldSpace_fragmentPosition;
35
36     gl_Position = projection * view * model * vec4(position, 1.0);
37
38 }
```

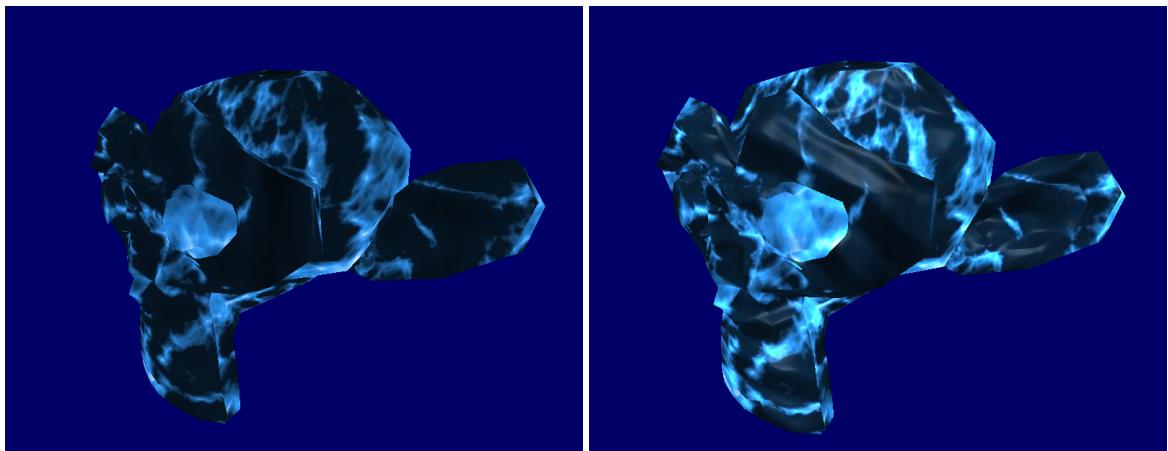
-Fragment Shader :

```
1      #version 330 core
2
3  out vec4 FragColor;
4
5  in vec3 worldSpace_fragmentPosition;
6  in vec2 o_uv0;
7  in vec3 tangentSpace_lightPosition;
8  in vec3 tangentSpace_viewPosition;
9  in vec3 tangentSpace_fragmentPosition;
10
11 uniform sampler2D colorTexture;
12 uniform sampler2D normalTexture;
13 uniform vec3 light_position;
14 uniform vec3 view_position;
15
16 void main() {
17     vec3 normale = texture(normalTexture, o_uv0).rgb;
18     normale = normalize(normale*2.0 - 1.0);
19
20     vec3 color = texture(colorTexture, o_uv0).rgb;
21
22     vec3 ambient = 1.0 * color;
23
24     vec3 light_direction = normalize(tangentSpace_lightPosition -
25                                         tangentSpace_fragmentPosition);
26     float diff = max(dot(light_direction, normale), 0.0);
27     vec3 diffuse = diff*color;
28
29     vec3 view_direction = normalize(tangentSpace_viewPosition - tangentSpace_fragmentPosition)
30     ;
31     vec3 reflexion = reflect(-light_direction, normale);
32     vec3 halfway = normalize(light_direction + view_direction);
33     float spec = pow(max(dot(normale, halfway), 0.0), 32.0);
34     vec3 specular = vec3(0.2)*spec;
35
36     FragColor = vec4(ambient + diffuse + specular, 1.0);
}
```

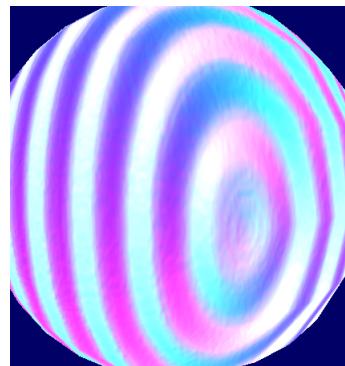
Une fois cette partie faite, le reste se déroule exactement de la même manière que pour les textures 2D. J'ai ainsi obtenu les résultats suivants :



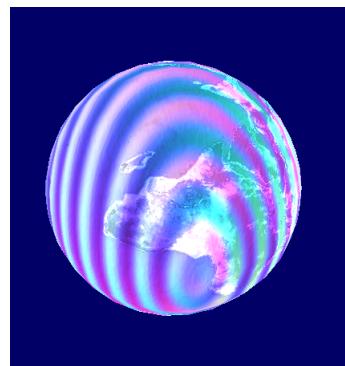
Et pour mieux voir, ci-dessous un avant-après de suzanne :



Durant cette partie du tp j'ai quand même fait face à plusieurs problèmes, le premier étant le fait que ma normal map prenait le dessus sur ma texture de base, comme vous pouvez le voir juste en dessous :



J'ai ensuite compris que le problème venait de l'indexage des textures. En effet, je ne les avait pas indexées correctement et donc forcément la seconde prenait le dessus sur la première. Puis j'ai eu un autre soucis. J'avais à la fois ma texture de base, mais celle-ci prenait la couleur de ma normal map :



Pour régler ce problème, j'ai alors changé l'ordre dans lequel j'utilisais ma texture, c'est à dire que j'ai fait : d'abord l'activation de la texture, ensuite la liaison et enfin je récupère la location. Autrement ça ne fonctionnait pas.

Au cours de ce niveau, j'ai appris à programmer Phong en passant par les shaders, à ajouter une normale map afin d'obtenir du relief, et aussi à comprendre l'importance de l'indexage des textures.

II.3 Niveau 2 : skybox

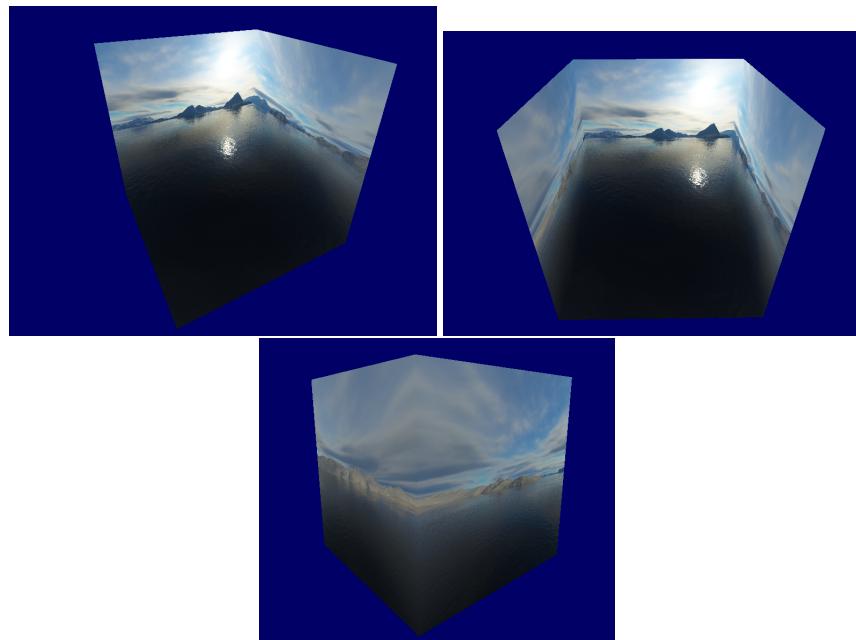
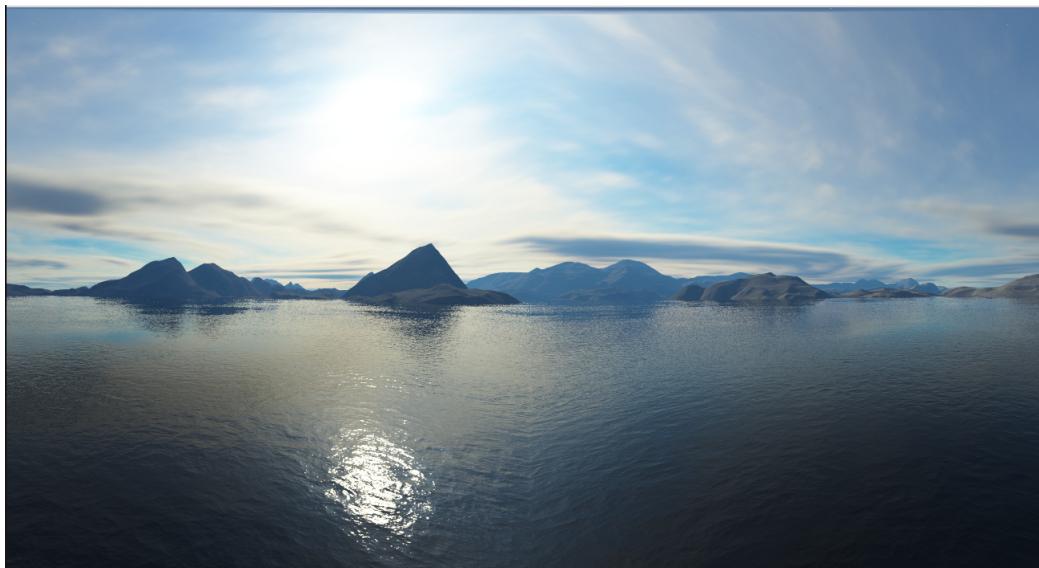
Dans cette partie, le but était de plaquer des textures sur les faces intérieures d'un cube et de placer notre caméra à l'intérieur de ce cube afin de donner une impression d'immensité de paysage.

Pour faire ceci, j'ai décidé de générer un cube moi même avec un tableau de sommets, puis de faire toutes les étapes nécessaires (bindVertexArray, bindBuffers...) pour le dessiner. Une fois le cube obtenu, le plaquage de 6 textures à la fois se fait de la même manière que pour les textures 2D, à quelques exceptions près. En effet, la manière de charger les textures n'est pas la même, elle est très similaire certes, mais il y a quelques différences comme vous pouvez le voir ci-dessous :

```
1     GLuint loadSkybox( std :: vector<std :: string > faces){
2
3     GLuint texture;
4
5     glGenTextures(1, &texture);
6     glBindTexture(GL_TEXTURE_CUBE_MAP, texture);
7
8     int width, height, nrChannels;
9     unsigned char * data;
10
11    for(unsigned int i=0; i<faces.size(); i++){
12
13        data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels, 0);
14
15        if(data){
16
17            stbi_set_flip_vertically_on_load(false);
18
19            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0,
20            GL_RGB, GL_UNSIGNED_BYTE, data);
21
22            stbi_image_free(data);
23
24        }
25
26        else{
27
28            std :: cout<<"ERROR!"<<std :: endl;
29
30            stbi_image_free(data);
31
32        }
33
34    }
35
36    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
37    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
38    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
39    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
40    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
41
42}
```

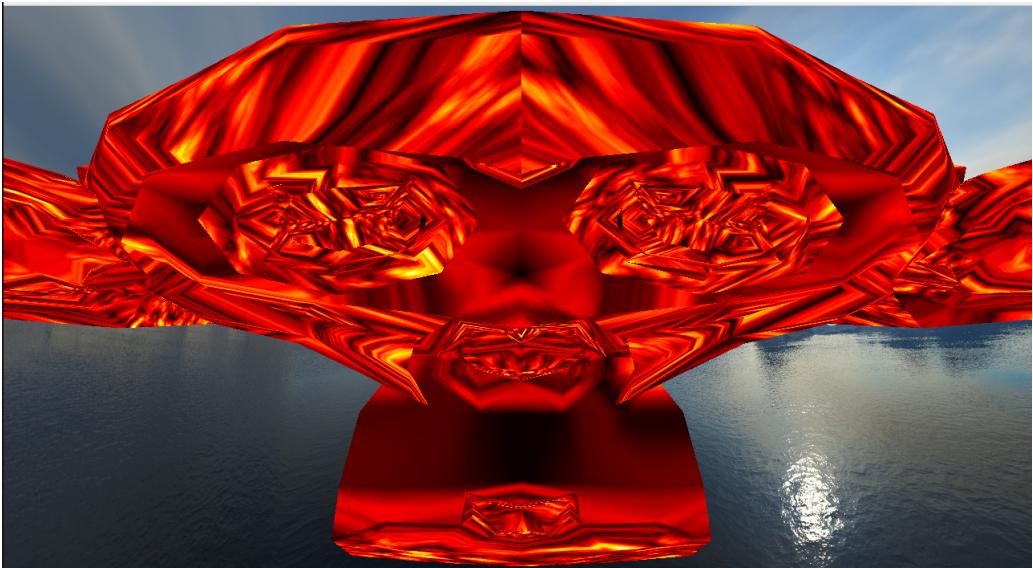
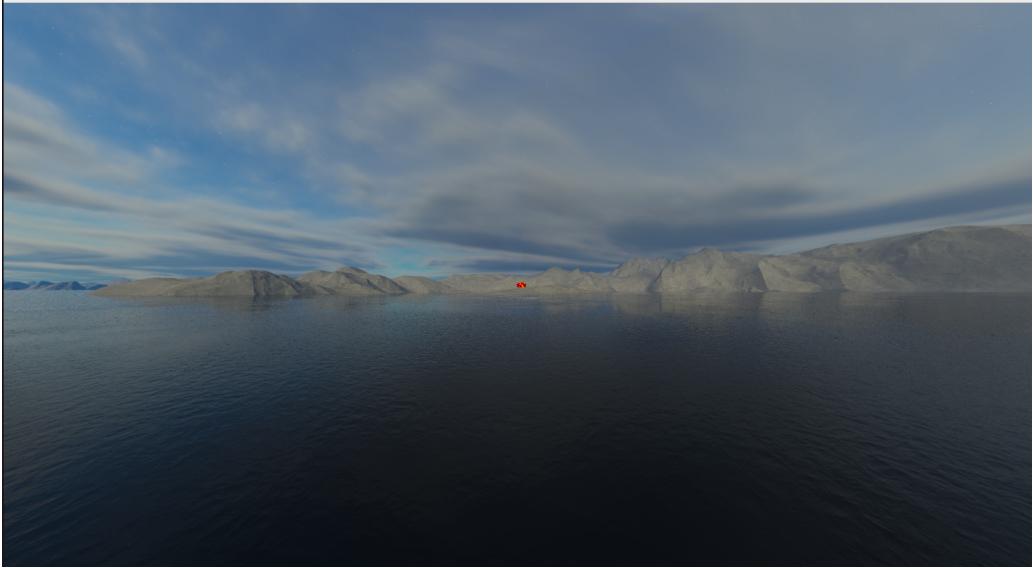
```
26     return texture;  
27 }
```

En effet, il existe en fait un type OpenGL spécifiquement adapté pour les cubemaps, ce qui nous facilite énormément la tâche. Autre petite modification à faire est dans le fragment shader dans lequel la variable associée à la texture n'est plus un sampler2D mais un samplerCube. Vous pouvez voir ci-dessous le résultat escompté :



Cependant, j'ai dû faire face à un problème qui était celui de pouvoir mettre un modèle 3D dans cette skybox. Après plusieurs tentatives, j'ai compris qu'il fallait dissocier l'apparition de la skybox, de la génération du mesh. C'est pour cela que j'ai créé un entier prenant 0 ou 1 afin de savoir quoi afficher, quand et avec quel contexte.

Mais la chose la plus important à comprendre est qu'il faut absolument créer deux nouveaux shaders pour chaque nouvelle instance ajoutée dans la scène. C'est comme ça que j'ai obtenu ce que vous pouvez voir dessous :



Cependant, comme vous pouvez le voir, un gros problème subsiste qui est le fait que en fonction du modèle 3D choisis, celui-ci aura une position et une taille bien définie. D'où le fait que suzanne soit très zoomée et que la boom box soit très petite. Normalement avec le code effectué, je devrais pourvoir zoomer et dézoomer mes modèles afin qu'il prennent une place cohérente dans la skybox. Mais j'ai trouvé d'où venait le soucis, il venait du fait que dans mes shaders en rapport avec le mesh je n'avais pas mis de matrices view, projection et model.

Maintenant que nous avons un modèle qui s'affiche correctement, nous allons tenter de faire en sorte qu'il fasse de la réflexion. Pour ceci, il a fallut modifier légèrement les shaders ainsi qu'une petite partie du code. En effet, l'utilisation de la fonction `reflect()` de GLSL était nécessaire :

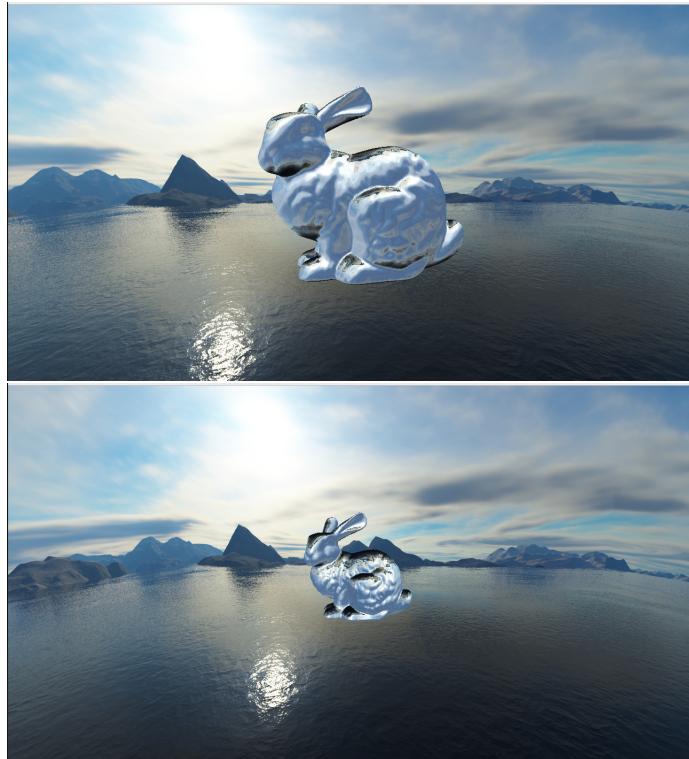
-Fragment shader :

```
1      #version 330 core
2  out vec4 FragColor;
3
4  in vec3 Normal;
5  in vec3 position;
6
7  uniform vec3 cam_pos;
8  uniform samplerCube skybox;
9
10 void main()
11 {
12     vec3 I = normalize(position - cam_pos);
13     vec3 R = reflect(I, normalize(Normal));
14     FragColor = vec4(texture(skybox, R).rgb, 1.0);
15 }
```

-Vertex shader :

```
1      layout (location = 0) in vec3 aPos;
2 // layout (location = 1) in vec3 aColor;
3 // layout (location = 2) in vec2 aTexCoord;
4 layout(location=1) in vec3 normal;
5
6 // out vec3 ourColor;
7 // out vec2 TexCoord;
8
9 out vec3 Normal;
10 out vec3 position;
11
12 uniform mat4 model;
13 uniform mat4 view;
14 uniform mat4 projection;
15
16 void main()
17 {
18     Normal = mat3(transpose(inverse(model)))*normal;
19     position = vec3(model*vec4(aPos, 1.0));
20     gl_Position = projection * view * model * vec4(aPos, 1.0);
21 }
```

Comme vous pouvez le voir ci-dessous, j'ai obtenu quelque chose de plutôt cohérent puisque le lapin est très réfléchissant. Cependant, aux vues d'autres exemples, je doute que le résultat soit réellement celui attendu, car il me semble que la réflexion n'est pas la bonne sur le dessus du lapin :



Par manque de temps et voulant essayer de faire le PBR, je me suis arrêtée là pour cette partie.

Au cours de cette partie, j'ai appris comment créer une cubemap, mais aussi comment lier plusieurs shader à mon programme afin de pouvoir manipuler plusieurs instances à la fois.

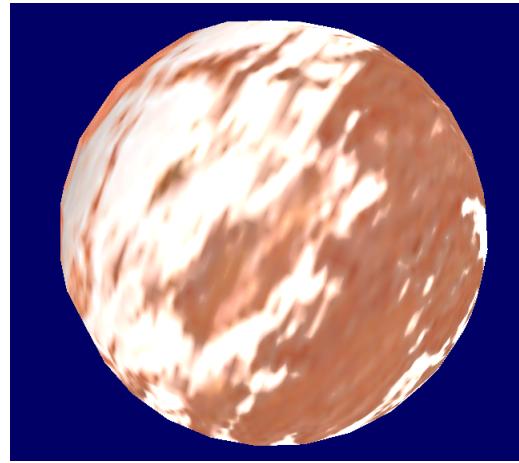
III TP9 :

III.1 Niveau 0 :

Dans ce premier niveau, le but était d'arriver à plaquer plusieurs textures et à manipuler la lumière de telle sorte que notre rendu soit le plus réaliste possible, car basé sur la physique.

Tout d'abord, il y a la texture correspondant à la couleur de base, que nous avons déjà réussi à faire dans le niveau 0 de notre TP8. Ensuite, il y a la texture normal map, que nous avons effectué lors du niveau 1 du TP8.

Donc, je me suis dit que pour commencer PBR, il serait intéressant de reproduire tout d'abord ce qui a été fait avant pour les deux premières textures :



Une fois ceci fait j'ai mis en application le cours avec l'aide de learnOpenGL, en codant les 3 fonctions nécessaires pour avoir un résultat correct : *distributionNormale()*, *geometrique()*, *fresnel()*. Ces trois fonctions sont indispensables au calcul de la réflectance. Elle font partie intégrante de l'équation :

```

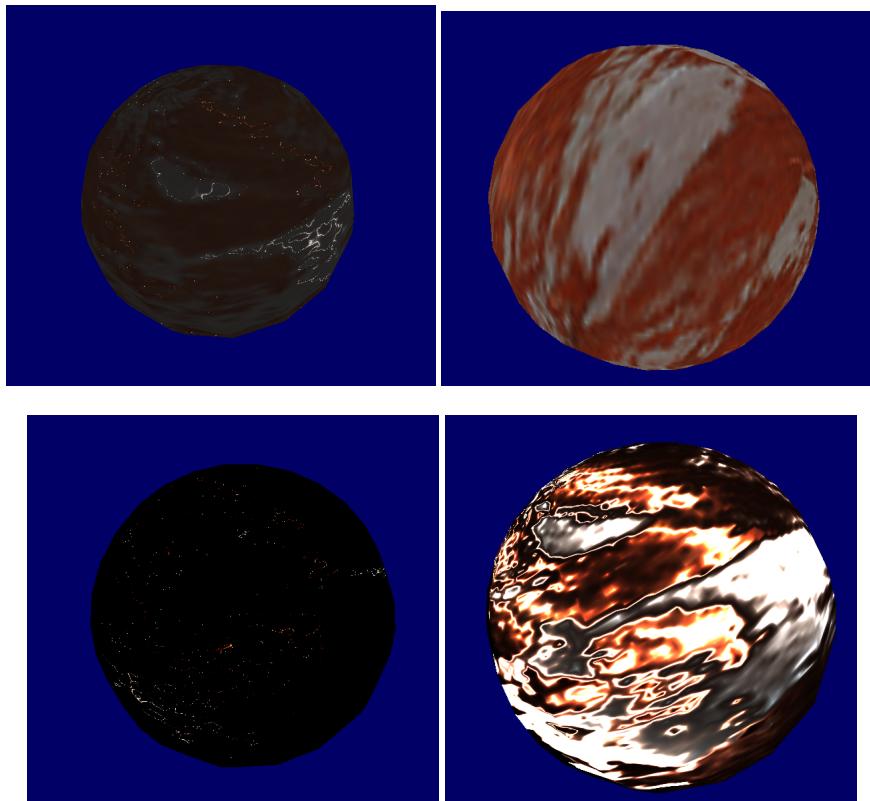
1   // distribution normale
2   float distribution_normale(vec3 N, vec3 H, float rugosite){
3       float a = rugosite*rugosite;
4       float a2 = a*a;
5       float NH = max(dot(N,H), 0.0);
6       float NH2 = NH*NH;
7
8       float numerateur = a2;
9       float denominateur = (NH2 * (a2 - 1.0) + 1.0);
10      denominateur = PI * denominateur * denominateur;
11
12      return numerateur/denominateur;
13  }
14
15  // Geometric Schlick function
16  float geometrie_schlick_beckmann(float NV, float rugosite){
17      float r = (rugosite + 1.0);
18      float k = (r * r) / 8.0;
19
20      float numerateur = NV;
21      float denominateur = NV * (1.0 - k) + k;
22
23      return numerateur/denominateur;
24  }

```

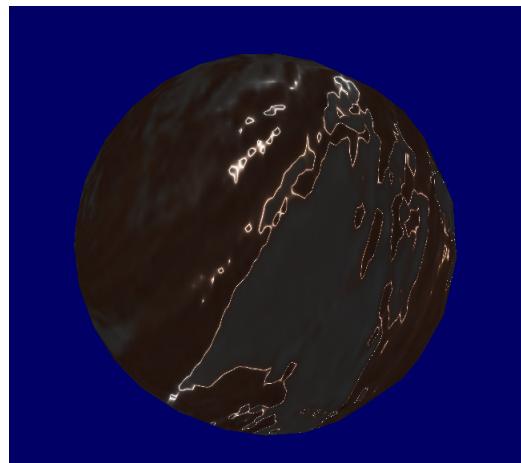
```

25
26 // Geometric Smith function that uses Schlick function
27 float geometrie_smith(vec3 N, vec3 V, vec3 L, float rugosite){
28     float NV = max(dot(N,V), 0.0);
29     float NL = max(dot(N,L), 0.0);
30     float ggx2 = geometrie_schlick_beckmann(NV, rugosite);
31     float ggx1 = geometrie_schlick_beckmann(NL, rugosite);
32
33     return ggx1*ggx2;
34 }
35
36 // Fresnel function
37 vec3 fresnel(float costeta, vec3 F0){
38     return F0+(1.0-F0)*pow(clamp(1.0 - costeta, 0.0, 1.0), 5.0);
39 }
```

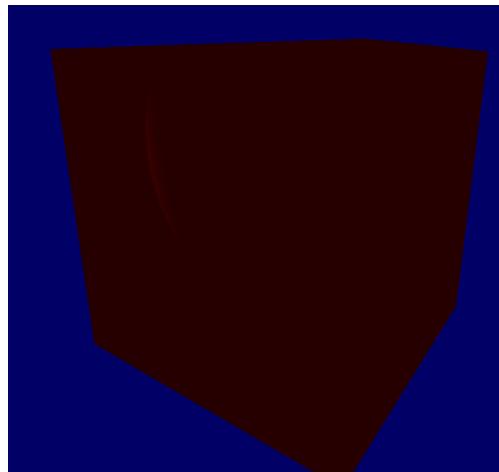
Une fois ces fonctions obtenues, de nombreux calculs sont encore à effectuer afin d'obtenir la couleur finale de notre objet. Celle-ci prendra en compte les textures concernant l'aspect métallique, rugueux, mais aussi l'occlusion ambiante. J'ai obtenu plusieurs résultats intermédiaires :



Pour arriver au résultat final :



Comme vous pouvez le constater, je n'ai pas le bon résultat. J'ai donc décidé de tenter une autre méthode qui cette fois n'utilise pas de carte de textures. Cette fois elle utilise des coefficients choisis donnant la pourcentage de tel texture sur l'objet. Les calculs sont sensiblement les même, mais le résultat n'a rien à voir avec le précédent :



Et vous pouvez également le voir, ce n'est pas la bon résultat.

Durant cette partie, j'ai appris à quoi correspondait un rendu basé sur la physique, et comment toutes les variables entrant en jeux dans la physique se calculaient.

Par manque de temps je n'ai pas pu chercher plus de solutions, et n'ai donc pas non plus pu finir le TP9.