LIRZIN
Léo
M1 IMAGINE
22200823

HAI719I – Programmation 3D
TP8 – Textures

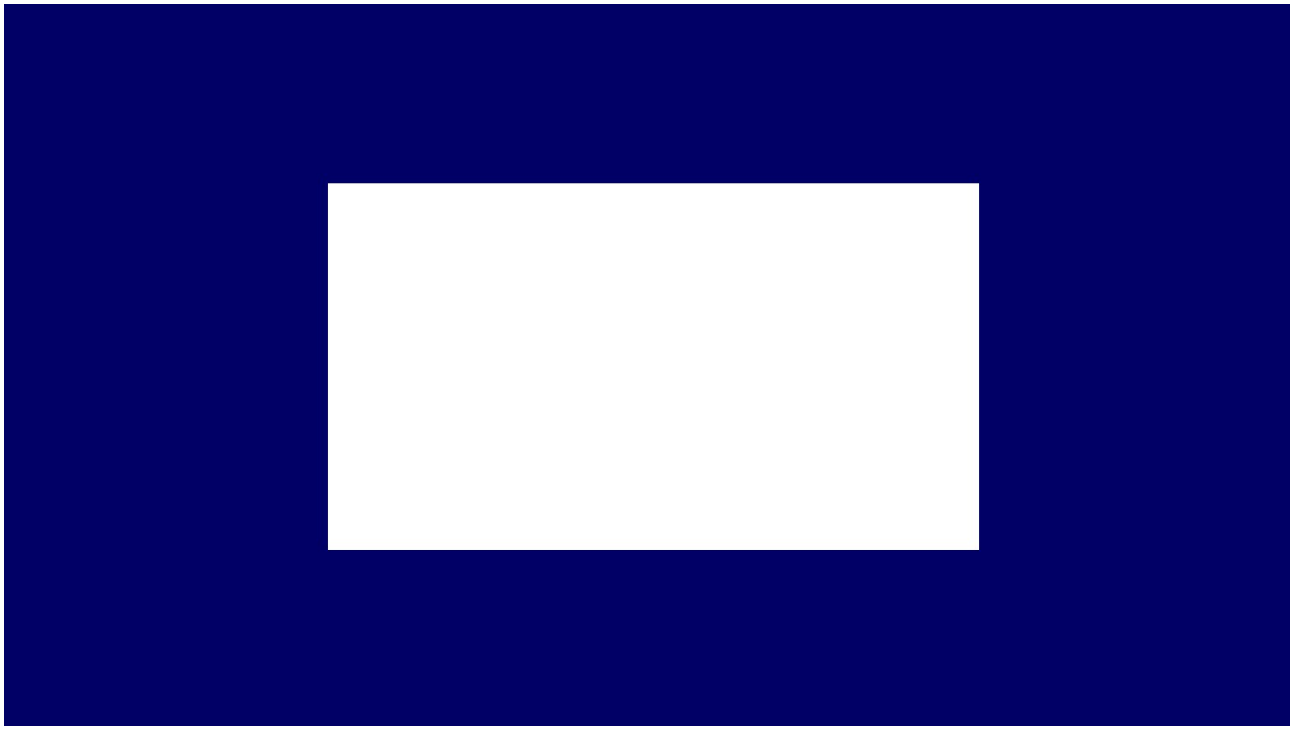
Niveau 0 :

Le niveau 0 du TP8 est implémenté dans le dossier N0TP8. Le reste du TP8 est implémenté dans le dossier N12TP8.

1) Créer un plan 3D

On créé un plan 3D en affichant deux triangles rectangles. Dans notre cas, les coordonnées de leurs sommets sont ((-0.5, -0.5, 0), (0.5, -0.5, 0), (0.5, 0.5, 0)) et ((-0.5, -0.5, 0), (-0.5, 0.5, 0), (0.5, 0.5, 0)). Pour afficher ces triangles, on appelle :
- glGenBuffers, glBindBuffer et glBufferData pour créer un buffer pour les sommets, le binder et envoyer les données des sommets au GPU (respectivement).
- glBindBuffer, glVertexAttribPointer et glEnableVertexAttribArray pour binder et activer le tableau de sommets donné au GPU.
- glDrawArrays pour dessiner les triangles sachant que le tableau donné au GPU est un tableau de coordonnées de points.

On obtient ce rendu là :

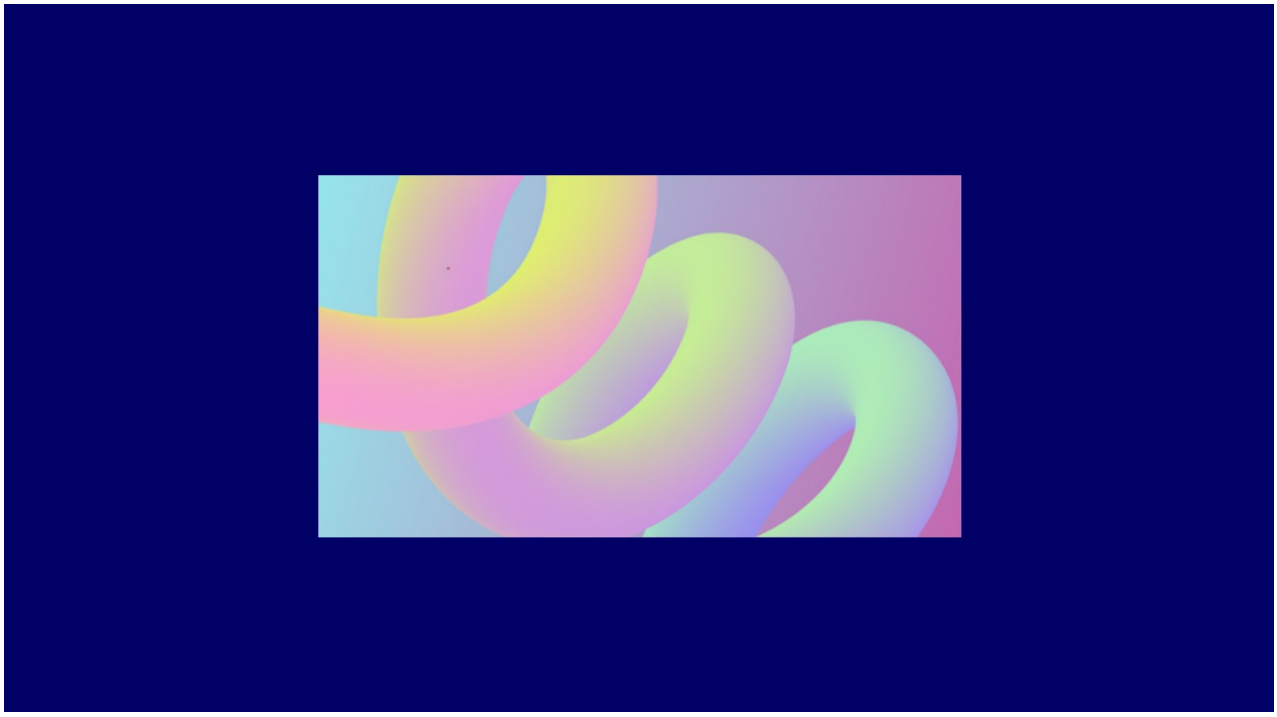


2) Charger une texture 2D

On charge une texture 2D à l'aide de la librairie stb_image.
On crée d'abord une texture en appelant glGenTextures(1, &*texture*), où *texture* est notre identifiant de texture, qu'on bind ensuite avec glBindTexture(GL_TEXTURE_2D, *texture*).
Pour charger la texture, on appelle stbi_load() qui renvoyer des données à ensuite donner au GPU. Cette fonction renseigne aussi la hauteur, largeur et le nombre de composants par pixel. Si ce nombre est égal à 3 alors chaque pixel est encodé RGB. Sinon s'il est égal à 4 alors chaque pixel est encodé RGBA.
Si la fonction retourne des données, on génère une texture 2D en appelant glTexImage2D(GL_TEXTURE_2D, 0, GL_{RGB|RGBA}, [hauteur de la texture], [largeur de la texture], 0, GL_{RGB|RGBA}, GL_UNSIGNED_BYTE, [données récupérées]) et glGenerateMipmap(GL_TEXTURE_2D). Il ne faut pas oublier de libérer les données renvoyées par stbi_load avec stbi_image_free([données récupérées]).
Bien sur, il faut pas oublier de désallouer la texture en appelant glDeleteTextures().
Pour envoyer la texture aux shaders, on appelle d'abord glEnable(GL_TEXTURE_2D) pour pouvoir envoyer une texture. Puis, on appelle glBindTexture(GL_TEXTURE_2D, [texture à envoyer]) et glActiveTexture(GL_TEXTURE0) pour mettre la texture en buffer puis l'activer.
Pour finir, on appelle glUniformli(glGetUniformLocation(progID, (char*) [nom de la texture dans le fragment shader], GL_TEXTURE0) pour passer la texture au fragment shader.

3) Plaquer la texture 2D sur le plan 3D

Avant d'envoyer la texture au fragment shader, il faut définit des uv pour chaque triangle à dessiner, c'est-à-dire un duo de valeurs pour chaque sommet. Cela servira à définir où sont les sommets par rapport à la texture que l'on veut afficher sur les triangles. Dans notre cas, les uv sont définis tels que ((1, 1), (0, 1), (0, 0)) et ((1, 1), (1, 0), (0, 0)). Une fois le tableau de uv défini, on peut l'envoyer aux shaders en créant un buffer, en bindant ce tableau au buffer et en envoyant les données dans le buffer au GPU. Finalement, on applique la texture sur les triangles dans le fragment shader en appelant texture(<sampler2D> [texture passé en uniforme], <vec2> [uv passé en in]) qui retourne un vec4 décrivant le texel aux coordonnées (u, v) dans la texture passée en uniform. Voici le rendu final de cette partie, où on affiche une image sur deux triangles formant un quad :



Niveau 1 :

1) Charger une texture en niveau de gris

Le chargement d'une bumpmap est très similair au chargement d'une texture dans le niveau 0. Toutes les fonctions à appeler pour le bon chargement d'une texture sont regroupées dans une fonction appelée loadTexture2DFromFilePath() définie à la ligne 38 dans le fichier Texture.cpp. Elle prend en entrée un std::string décrivant le chemin relatif à l'endroit de l'éxecution vers la texture à charger et renvoie un ID de texture si la texture a bien été chargé. S'il y a eu un problème, une exception est lancée.
Dans la structure Material (dans le fichier Material.h), on rajoute un membre *m_bumpmap* de type GLint qui stocke l'identifiant de la texture indiquant une bumpmap. Par conséquent, on modifie la méthode Material::init() (dans le fichier Material.cpp à la ligne 24) pour faire en sorte que *m_bumpmap* ait un identifiant vers une texture en y stockant le résultat retourné par loadTexture2DFromFilePath([chemin relatif vers le bumpmap]). Bien évidemment, on rajoute un appel à glDeleteTexture(1, (GLuint*) &*m_bumpmap*) dans la méthode Material::clear() (ligne 42).
Pour envoyer le bumpmap au fragment shader, on procède de la même méthode que dans le niveau 0, c'est-à-dire en appelant glActiveTexture, glBindTexture et glUniformli. Seulement ici, on utilise GL_TEXTURE1 car GL_TEXTURE0 est déjà pris par la texture que l'on veut afficher sur notre mesh.

2) Utiliser cette texture pour modifier les normales du modèle

Je n'ai pas réussi à modifier les normales du mesh selon le bumpmap. Cependant, j'ai appliqué les formules du cours 1 suivantes dans le fragment shader :

Théorie de la perturbation des normales

On ne modifie pas la position du sommet P, mais la normale interpolée par

- Pentes
  - $d_u = \frac{\Delta z}{\Delta x} = \frac{s}{2}[h(i+1,j) - h(i-1,j)]$
  - $d_v = \frac{\Delta z}{\Delta y} = \frac{s}{2}[h(i,j+1) - h(i,j-1)]$

  où s est un facteur multiplicatif de la hauteur maximale en fonction de la taille d'un seul texel, $h(i,j)$ correspond aux texels de la texture.
- Calcul des directions dir_u, dir_v
  - $dir_u = (1, 0, d_u)$ et $dir_v = (0, 1, d_v)$
- Calcul de la nouvelle normale
  - $m = normalize(dir_u \times dir_v) = \frac{(-d_u, -d_v, 1)}{\sqrt{d_u^2 + d_v^2 + 1}}$

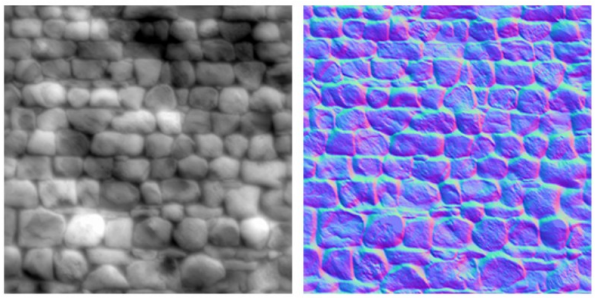
  où m est une normale dans l'espace tangent.

Sébastien Beugnon Textures 24/31

Alternative : Normal Mapping

Nouvelle forme de bump mapping

- Les calculs sont stockées directement dans l'image (baking)
- Attention : stockée dans l'espace tangent et normalisée
- $RGB = 0.5 * m + 0.5$



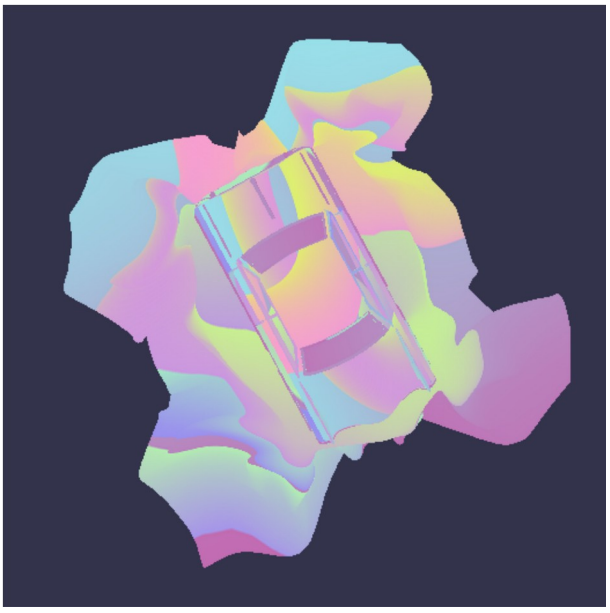
Ces formules sont traduites dans le fragment shader de la façon suivante :

```
float du = ( texture(bumpMap, vec2(o_uv0.x+0.01, o_uv0.y)).x - texture(bumpMap, vec2(o_uv0.x-0.01, o_uv0.y)).x )/2;
float dv = ( texture(bumpMap, vec2(o_uv0.x, o_uv0.y+0.01)).x - texture(bumpMap, vec2(o_uv0.x, o_uv0.y-0.01)).x )/2;
vec3 diru = vec3(1, 0, du);
vec3 dirv = vec3(0, 1, dv);
vec3 m = diru*dirv;
FragColor = vec4( 0.5*normalize(m) + 0.5, 1.0 ) * texture(colorTexture, o_uv0.xy);// * color;
```

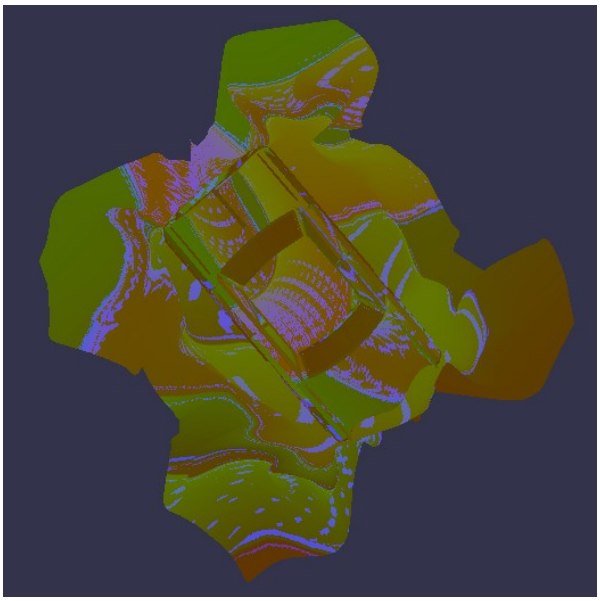
Pour afficher la texture simple, on utilise cette ligne de code dans le fragment shader :

```
FragColor = texture(colorTexture, o_uv0.xy) * color;
```

Voici donc les rendus avec et sans bumpmap, pour comparer les résultats :



Sans bumpmapping



Avec bumpmapping

On peut clairement voir que le résultat obtenu n'est pas celui attendu.

Niveau 2 :

1) Créer un cube 3D pour la skybox

Le cube 3D pour la skybox est défini et affiché dans Mesh::draw_skybox() (dans le fichier Mesh.cpp à la ligne 32). Dedans, les sommets du cube sont déclarées dans un tableau de flottants. On mutliplie chaque coordonnées par un grand nombre pour que le cube devienne une skybox. Ensuite, on crée un VertexBufferObject et un VertexArrayObject à partir du tableau des sommets. Finalement, on envoie au GPU ce tableau et on dessine le cube. Pour l'instant, on ne change rien au niveau des shaders.

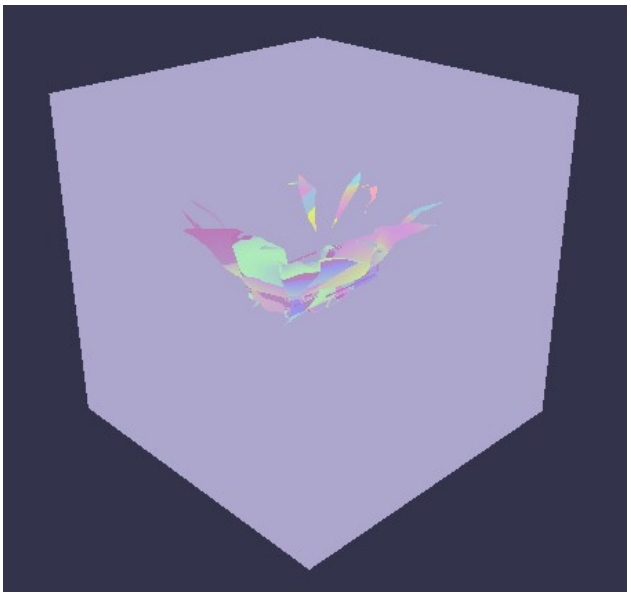
```
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
layout(location = 2) in vec3 tangent;
layout(location = 3) in vec3 uv0;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
//uniform mat4 mvp;
//uniform mat4 modelView;
//uniform mat4 normalMatrix;


out vec3 o_positionWorld;
out vec3 o_normalWorld;
out vec3 o_uv0;

void main() {
  mat3 normalMatrix = mat3(transpose(inverse(model)));
  o_uv0 = uv0;
  vec4 positionWorld = model * vec4(position, 1.0);
  o_positionWorld = positionWorld.xyz;

  o_normalWorld = normalMatrix * normal;
  gl_Position = projection * view * positionWorld;
}
```

Code du vertex shader

```
in vec3 o_positionWorld;
in vec3 o_normalWorld;
in vec3 o_uv0;
out vec4 FragColor;

uniform vec3 camera_pos;
uniform vec4 color;
uniform sampler2D colorTexture;
uniform sampler2D bumpMap;
uniform samplerCube skybox;

void main()
{
    FragColor = texture(colorTexture, o_uv0.xy) * color;
}
```

Code du fragment shader

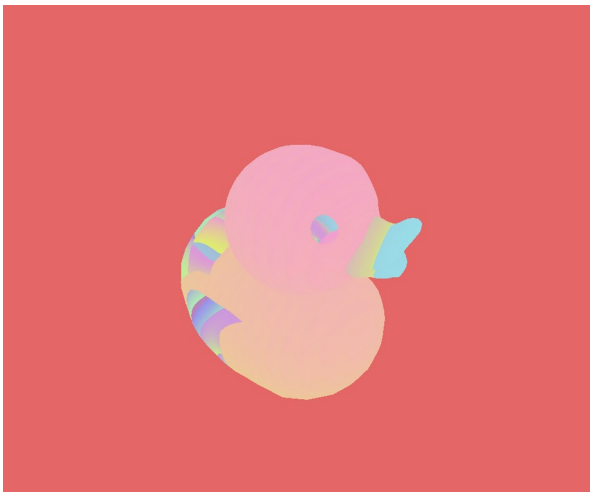
Voici les rendus obtenus grâce à cette partie et grâce aux shaders précedemment montrés. La texture utilisé ici est "gradient.png" :



modèle : ToyCar.glb

modèle : Duck.glb

On peut modifier le fragment shader pour modifier la couleur unie du cube. On remarque que si u et v sont proches de 1, alors on n'est pas en train de dessiner le modèle 3D :



Rendu obtenu grâce au code glsl à droite

```
in vec3 o_positionWorld;
in vec3 o_normalWorld;
in vec3 o_uv0;
out vec4 FragColor;

uniform vec3 camera_pos;
uniform vec4 color;
uniform sampler2D colorTexture;
uniform sampler2D bumpMap;
uniform samplerCube skybox;

void main()
{
    if( o_uv0.x > 0.999 && o_uv0.y > 0.999 )
    {
        FragColor = vec4(0.9, 0.4, 0.4, 1.);
    }
    else
    {
        FragColor = texture( colorTexture, o_uv0.xy ) * color;

    }
}
```

Nouveau code du fragment shader

2) Charger une texture cube

La fonction loadCubeMap() est définie à la ligne 9 dans le fichier Texture.cpp. Elle prend en entrée un std::vector de chemin vers des textures et retourne une ID de texture indiquant une texture cube. Cette fonction est identique à celle qui se trouve dans le cours. Pour indiquer que notre texture est bien une texture cube, on utilise GL_TEXTURE_CUBE_MAP quand on bind la texture.
Si tout se passe bien, la fonction retourne un ID que l'on stocke ensuite dans *m_skybox*. *m_skybox* est un nouveau membre de Material stockant l'ID de la texture cube. Bien sûr, elle est libérée dans clean avec glDeleteTexture. De plus, elle est activée avec glActiveTexture(GL_TEXTURE2) et bindée avec glBindTexture(GL_TEXTURE_CUBE_MAP, m_skybox). Enfin, elle est envoyée au fragment shader grâce à glUniformli.

3) Créer un shader faisant le rendu de la skybox

Deux shaders ont été créés pour rendre la skybox. Elle se situe dans le même dossier que le vertex shader et le fragment shader fournis avec la base de code.
Dans le vertex shader de la skybox nommé vertexskybox.glsl, on récupère la position monde dans la location = 0. Ces données sont renseignées dans la méthode Mesh::draw_skybox() précédemment mentionnée. Dans ce shader, on définit *gl_Position* comme la matrice MVP et on renseigne *TexCoord*, récupéré par le fragment shader. On lui donne la valeur de position transformé par la matrice *model*.
Dans le fragment shader de la skybox nomme fragmentskybox.glsl, on définit à partir de *TexCoord* le vecteur directeur caméra-position *I*. Finalement, la couleur du fragment est définie par texture(skybox, I) pour les composantes RGB et 1 pour la composante A.

Dans le code C++, on crée un nouveau programme en ajoutant un membre *m_skybox_program* de type GLuint dans la structure Material. Ce programme est initialisé en appelant load_shaders("shaders/unlit/vertexskybox.glsl", "shaders/unlit/fragmentskybox.glsl").
Utiliser deux program entrainent quelques modifications dans le code. Notamment, dans le destructeur de la structure Material où il faut rajouter un appel à glDeleteProgram(*m_skybox_program*). De plus, la signature de la méthode Material::bind() (à la ligne 58 dans le fichier Material.cpp). Cette méthode prend maintenant un argument nommé *m_cur_program* qui décrit le program à utiliser. Il y a donc un appel à glUseProgram(*m_cur_program*) dans cette méthode.
Malgrè mes efforts, les shaders de skybox ne semble pas marcher. Pour démontrer cela, on va montrer deux rendus obtenus en utilisant et en n'utilisant pas *m_skybox_program*, implémentant les shaders de skybox :

```
for (int i = 0; i < Context::instances.size(); ++i) {
    Instance& inst = Context::instances[i];
    Material* material = inst.material;
    Mesh* mesh = inst.mesh;

    material->bind(material->m_skybox_program);
    material->setMatrices(Context::camera.projection, Context::camera.view, inst.matrix);
    Mesh::draw_skybox();

    material->bind(material->m_program);
    material->setMatrices(Context::camera.projection, Context::camera.view, inst.matrix);
    mesh->draw();
}
```

Rendu de Duck.glb en implémentant les shaders de skybox. On s'aperçoit que la skybox n'est pas rendu à l'écran.

```
for (int i = 0; i < Context::instances.size(); ++i) {
    Instance& inst = Context::instances[i];
    Material* material = inst.material;
    Mesh* mesh = inst.mesh;

    //material->bind(material->m_skybox_program);
    material->setMatrices(Context::camera.projection, Context::camera.view, inst.matrix);
    Mesh::draw_skybox();

    material->bind(material->m_program);
    material->setMatrices(Context::camera.projection, Context::camera.view, inst.matrix);
    mesh->draw();
}
```

Rendu de Duck.glb en n'implémentant pas les shaders de skybox. On s'aperçoit que la skybox est bien rendu à l'écran comme il faut.

4) Faire un premier rendu de la skybox, faire un rendu réfléctif d'un objet 3D

Pour rendre la skybox, il faut changer le fragment shader.

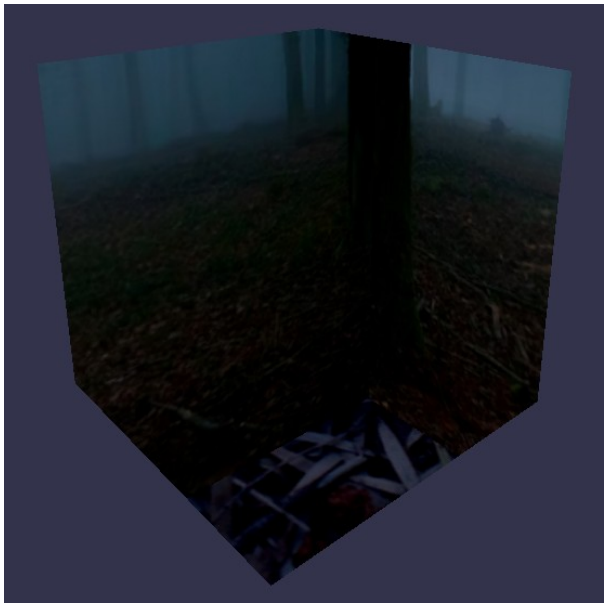
I correspond au vecteur direction de la camera vers la position monde.

Vu qu'une skybox est dessiné en 3 dimensions, texture ne prend pas un vec2 mais un vec3. Dans notre cas, ce vec3 est I.

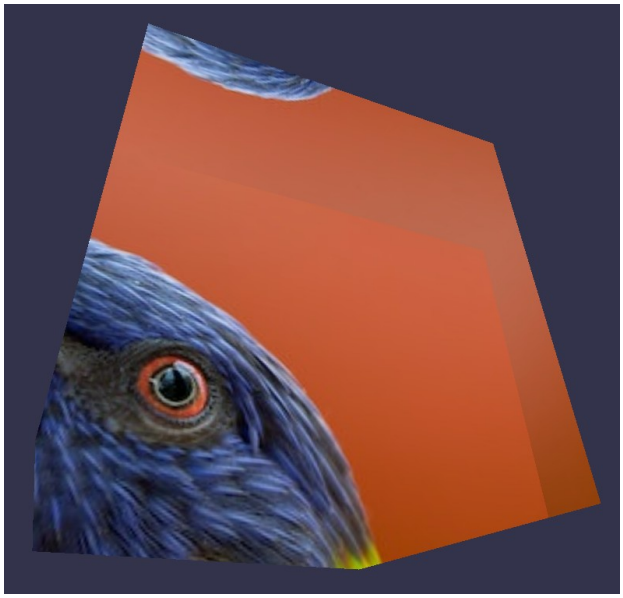
Pour ces rendus, on a rétréci la taille du cube.

```
in vec3 o_positionWorld;
in vec3 o_normalWorld;
in vec3 o_uv0;
out vec4 FragColor;

uniform vec3 camera_pos;
uniform vec4 color;
uniform sampler2D colorTexture;
uniform sampler2D bumpMap;
uniform samplerCube skybox;

void main()
{
    vec3 I = o_positionWorld - camera_pos;
    FragColor = vec4 ( texture ( skybox, I ).rgb , 1.0);
}
```

Code glsl du fragment shader permettant de n'afficher que la skybox



*Skybox faites avec les photos prises dans les bois "*_skybox.png"*



skybox faite avec la photo du perroquet "pic.png"

Pour rendre le modèle 3D tel qu'il réfléchisse la lumière, on change le code du fragment shader tel que :

```
in vec3 o_positionWorld;
in vec3 o_normalWorld;
in vec3 o_uv0;
out vec4 FragColor;

uniform vec3 camera_pos;
uniform vec4 color;
uniform sampler2D colorTexture;
uniform sampler2D bumpMap;
uniform samplerCube skybox;

void main()
{
    vec3 N = normalize(o_normalWorld);
    vec3 I = o_positionWorld - camera_pos;
    vec3 R = reflect ( I, N );
    FragColor = vec4 ( texture ( skybox, R ).rgb , 1.0);
}
```

Cette fois, on appelle la fonction reflect() sur le vecteur direction caméra-position monde et la normale normalisée. En fait, elle calcule tout simplement **I – dot(I, N) * 2. * N**.

Le vec3 retourné est un nouveau rayon directeur décrivant la réfléction de *I* sur le modèle 3D tel que le triangle du modèle que *I* intersecte a pour normal *N*.

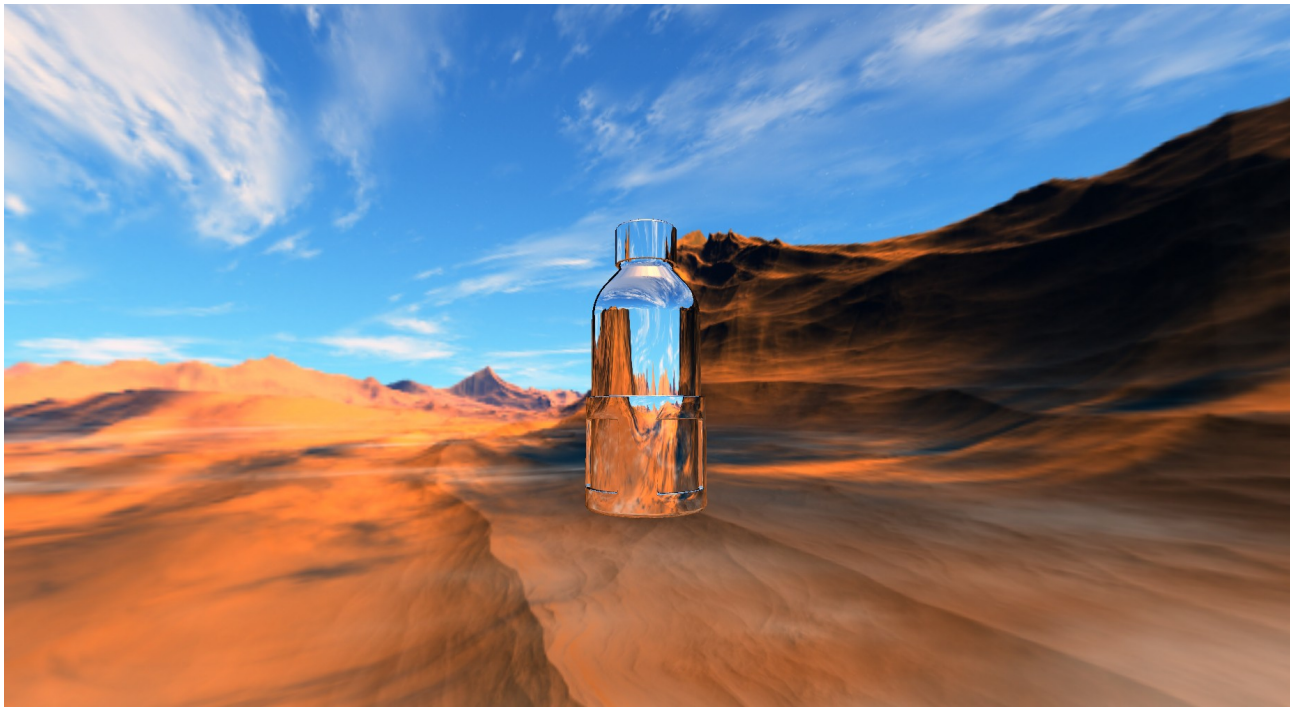
S'il y a aucune intersection, la skybox est dessiné. C'est-à-dire que reflect retourne *I*.

Voici les rendus finaux de ce TP :

Modèle ToyCar.glb réfléctif dans une boite englobante faite de perroquet "pic.png"

*Modèle Duck.glb réfléctif de face dans une boite englobante faite des photos "*_skybox.png"*

*Modèle Duck.glb réfléctif de face dans une boite englobante faite des photos "*_skybox.png"*



Modèle WaterBottle.glb réfléctif dans une boîte englobante faite des photos "sorbin_*.png"

LIRZIN
Léo
M1 IMAGINE
22200823

HAI719I – Programmation 3D
TP9 – Rendu PBR

Le TP9 est implémenté dans le dossier N0TP9.

Niveau 0 : Développer votre propre modèle PBR avec les paramètres suivants. Créer votre classe de matière associant un shader program et des paramètres à injecter dedans :

Toutes les modifications implémentant le niveau 0 ont été faites dans les fichiers Material.cpp et Material.h.
Pour chaque matière, il faut faire un nouveau shader. Dans le programme C++, on rajoute donc 5 programmes ce qu'il fait 6 programmes en tout :

#	Nom de la variable contenant l'ID du programme	Ce que fait le programme
0	*m_program*	Implémentation d'un fragment shader de base prenant en compte une texture avec ces uv seulement. Ce fragment shader s'appelle fragment.glsl
1	*m_phong_program*	Implémentation d'un fragment shader calculant les ombres selon le modèle de phong et la position d'une lumière. La couleur du modèle est unie. Ce fragment shader s'appelle fragmentPhong.glsl
2	*m_albedo_program*	Implémentation d'un fragment shader calculant les ombres selon le modèle de phong et la position d'une lumière. La couleur du modèle est dictée selon une texture. Ce fragment shader s'appelle fragmentAlbedo.glsl
3	*m_metalness_program*	Implémentation d'un fragment shader calculant les ombres selon le modèle de phong et la position d'une lumière. La couleur du modèle est dictée par une metalnessmap. Ce fragment shader s'appelle fragmentMetalness.glsl
4	*m_roughness_program*	Implémentation d'un fragment shader calculant les ombres selon le modèle de phong et la position d'une lumière. La couleur du modèle est dictée par une roughnessmap. Ce fragment shader s'appelle fragmentRoughness.glsl

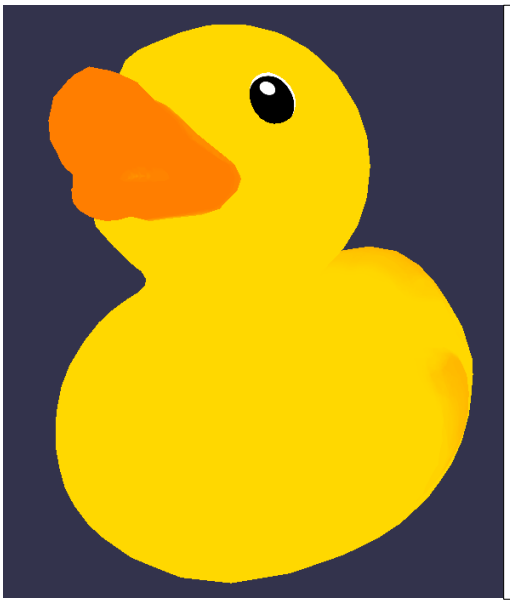
5 *m_AO_program* Implémentation d'un fragment shader calculant les ombres selon le modèle de phong et la position d'une lumière. En plus de ces ombres, on rajoute d'autres ombres se trouvant sur un AOmap.
Ce fragment shader s'appelle fragmentAO.glsl

Tous les shaders se trouve dans le dossier N012TP9/shaders/unlit.
Vous pouvez cycler entre ces différents programmes avec les touches '+' (pour avancer d'un numéro) et '-' (pour reculer d'un numéro).
Pour bien implémenter l'utilisation de différents programmes dans une même exécution, il a fallu ajouter *cur_material*, le numéro du matériau courant, *cur_program*, retenant l'ID du programme en train d'être utilisé et *nb_materials*, le nombre de matériaux implémentés pour que *cur_material* reste entre 0 et 5.

Pour implémenter le modèle de Phong, la position et la couleur de la lumière ont été renseignées dans la structure Material.
De plus, chaque matériau a ces propres composantes ambiente, diffuse, spéculaire et de brilliance.

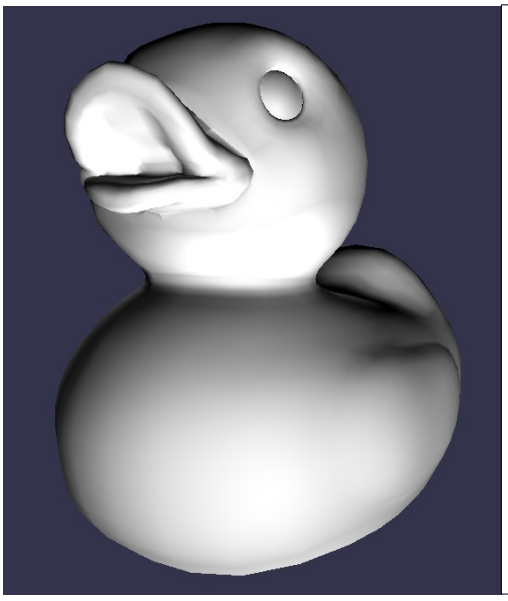
Le vertex shader commun à tous les programmes et la partie du fragment shader implémentant le modèle de Phong sont ceux présentés dans le deuxième cours aux pages 4 et 5.

Affichage du modèle WaterBottle.glb avec le programme *m_program* implémentant le fragment shader fragment.glsl



Affichage du modèle Duck.glb avec le programme *m_program* implémentant le fragment shader fragment.glsl

Affichage du modèle WaterBottle.glb avec le programme *m_phong_program* implémentant le fragment shader fragmentPhong.glsl



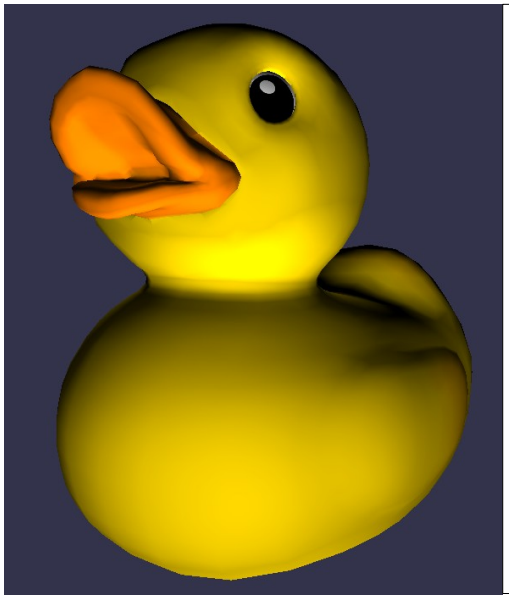
Affichage du modèle Duck.glb avec le programme *m_phong_program* implémentant le fragment shader fragmentPhong.glsl

Les paramètres pour les rendus de Phong sont : Ambient(0, 0, 0), Diffuse(1, 1, 1), Spéculaire(0.5, 0.5, 0.5), Brilliance = 1, PositionLumière(-2, 1, -1.5), CouleurLumière(1, 1, 1)

1) albedo

L'albedo est défini comme la couleur de base du matériau. La couleur de base du matériau est elle-même définie par une texture passée au fragment shader. La couleur finale est donc la couleur du texel de la texture indiqué par le couple (u, v) multiplié par la quantité de lumière que reçoit le modèle.

Affichage du modèle WaterBottle.glb avec le programme *m_albedo_program* implémentant le fragment shader fragmentAlbedo.glsl



Affichage du modèle Duck.glb avec le programme *m_albedo_program* implémentant le fragment shader fragmentAlbedo.glsl

Les paramètres pour les rendus d'albedo sont : Ambient(0, 0, 0), Diffuse(1, 1, 1), Spéculaire(0.5, 0.5, 0.5), Brilliance = 1, PositionLumière(-2, 1, -1.5), CouleurLumière(1, 1, 1)

2) metalness

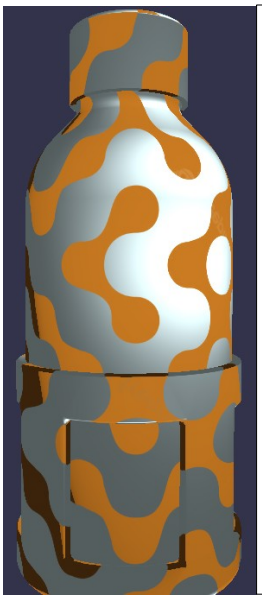
Le metalness est défini par une metalnessmap. Cette map est une texture en noir et blanc. Si le texel est blanc, alors le matériau affiché est métal. Si le texel est noir, alors le matériau affiché est diaéléctrique ou non-métal.
Cependant, même si une texture est noir et blanc, il existe des pixels gris entre les parties blanches et noires. C'est pour cela que la couleur est définie telle que :

CouleurFinale = R*CouleurMetal + (1.0 – R)*CouleurNonMetal
où R est la composante rouge du texel. Vu que la metalnessmap est une texture en noir et blanc, on considère les composantes R, G et B égales.

Les couleurs CouleurMetal et CouleurNonMetal sont définies à partir de composantes ambiante, diffuse, spéculaire et de brilliance. Ces composantes sont elle-mêmes définies dans Material.h. Ces dernières sont données aux fragment shader avant de dessiner le modèle.

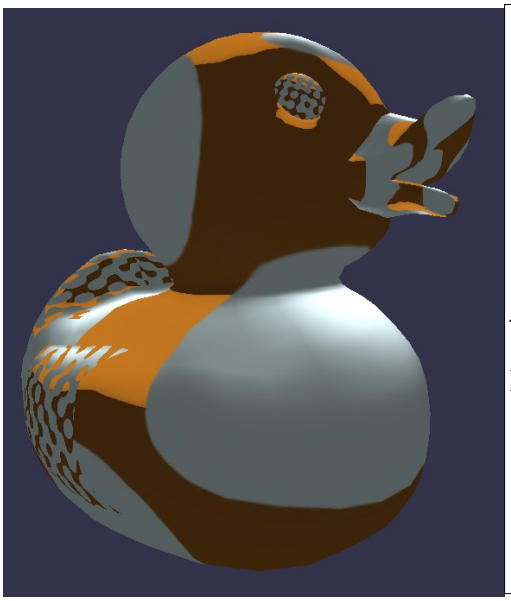
Pour déduire CouleurMetal, la quantité de lumière calculée (ambient + diffuse + speculère) est mutlipliée par une valeur obtenue grâce aux fonction de Fresnel. Cette valeur correspond à une couleur caractéristique d'un métal en particulier.

Pour déduire CouleurNonMetal, la quantité de lumière calculée (ambient + diffuse + speculère) est mutlipliée par une couleur obtenue depuis le code c++.



Affichage du modèle WaterBottle.glb avec le programme *m_metalness_program* implémentant le fragment shader FragmentMetalness.glsl

Metalmap utilisée : metalnessmap.png

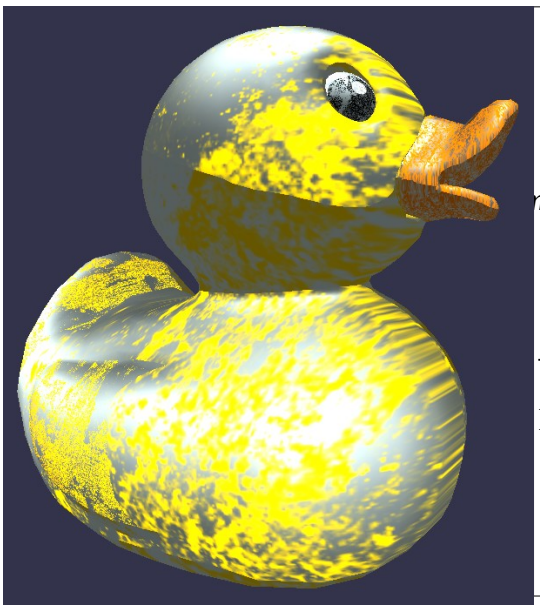


Affichage du modèle Duck.glb avec le programme *m_metalness_program* implémentant le fragment shader FragmentMetalness.glsl

Metalmap utilisée : metalnessmap.png

Affichage du modèle WaterBottle.glb avec le programme *m_metalness_program* implémentant le fragment shader FragmentMetalness.glsl

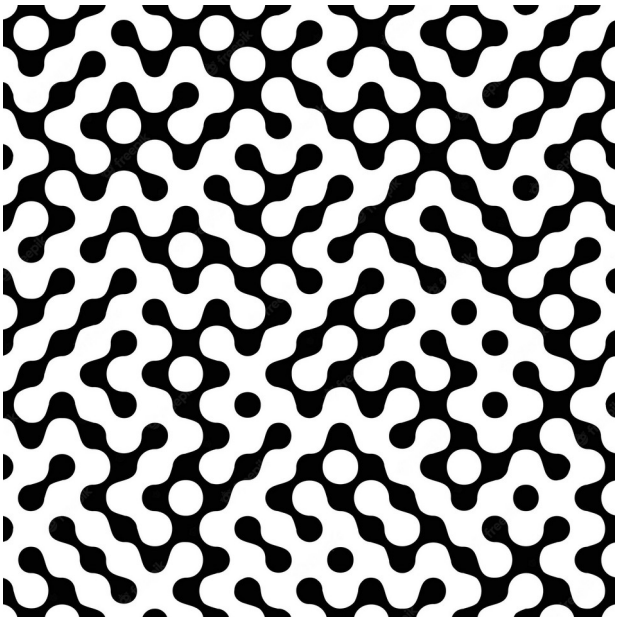
Metalmap utilisée : metalnessmap2.png



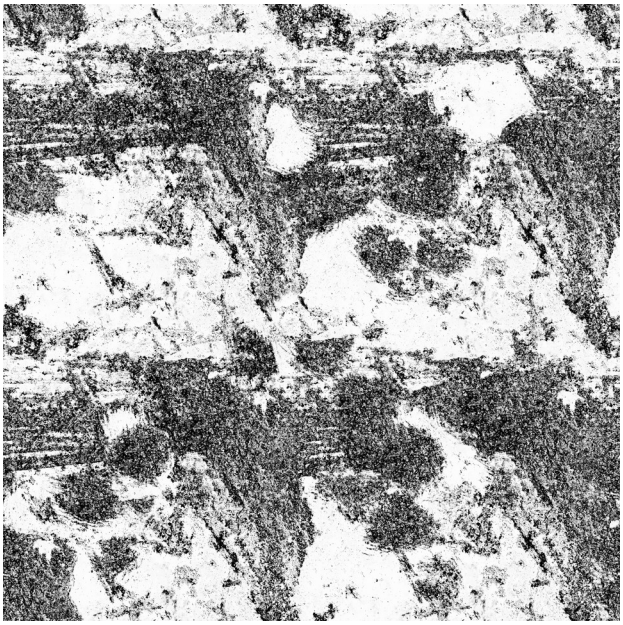
Affichage du modèle Duck.glb avec le programme

n_metalness_program implémentant le fragment shader FragmentMetalness.glsl

Metalmap utilisée : metalnessmap2.png



metalnessmap.png



metalnessmap2.png

Les paramètres pour la metal pour les rendus de Metalness sont : Ambient(0.4, 0.4, 0.4), Diffuse(0, 0, 0), Spéculaire(1, 1, 1), Brilliance = 3, ValeurFresnel(0.664, 0.824, 0.850) (Zinc)

Les paramètres pour la diéléctrique pour les rendus de Metalness utilisant metalnessmap.png sont : Ambient(0.4, 0.4, 0.4), Diffuse(0.05, 0.05, 0.05), Spéculaire(1, 1, 1), Brilliance = 0.1, CouleurBaseNonMetal(0.5661, 0.3412, 0.0902)

Dans le cas de WaterBottle.glb, la position de la lumière est (-2, 1, -1.5) tandis que dans le cas de Duck.glb la position de la lumière est (2, 1, 3). Dans les deux cas, la couleur de la lumière est (1, 1, 1).

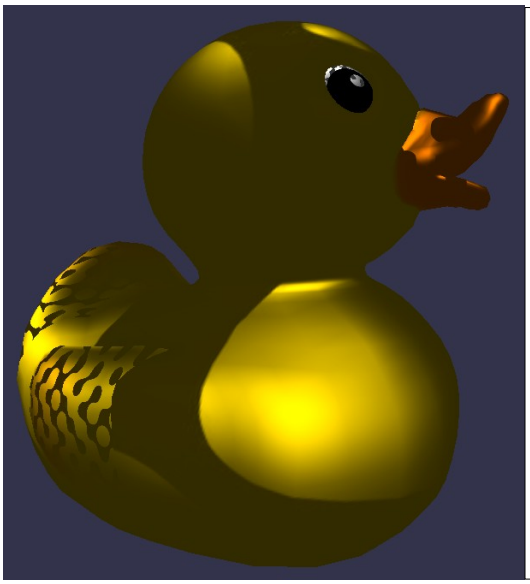
3) roughness

Le roughness est défini par une roughnessmap. Cette map est une texture en noir et blanc. Si le texel est blanc, alors le matériau affiché apparaît lisse. Si le texel est noir, alors le matériau affiché apparaît matte.
Cependant, même si une texture est noir et blanc, il existe des pixels gris entre les parties blanches et noires. C’est pour cela que la composante spéculaire est directement multipliée par la composante rouge du texel obtenu à partir de la roughnessmap avec les coordonnées (u, v). Vu que la roughnessmap est une texture en noir et blanc, on considère les composantes R, G et B égales.

Pour mettre en évidence les parties lisses et mattes, on met le niveau de brillance à 5.

Affichage du modèle WaterBottle.glb avec le programme *m_roughness_program* implémentant le fragment shader FragmentRoughness.glsl

Roughmap utilisée : metalnessmap.png

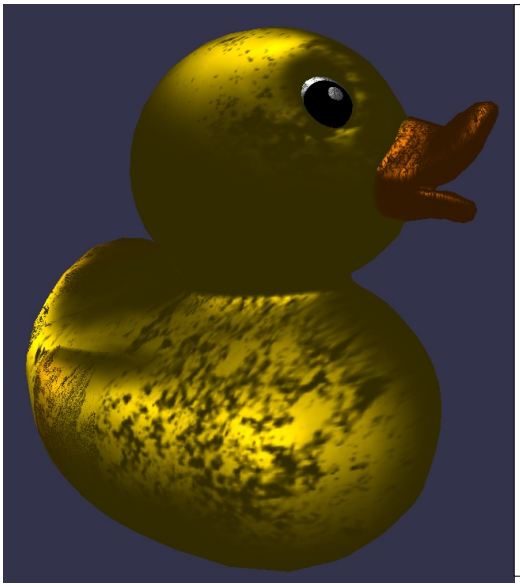


Affichage du modèle Duck.glb avec le programme *m_roughness_program* implémentant le fragment shader FragmentRoughness.glsl

Roughmap utilisée : metalnessmap.png

Affichage du modèle WaterBottle.glb avec le programme *m_roughness_program* implémentant le fragment shader FragmentRoughness.glsl

Roughmap utilisée : metalnessmap2.png



Affichage du modèle Duck.glb avec le programme *m_roughness_program* implémentant le fragment shader FragmentRoughness.glsl

Roughmap utilisée : metalnessmap2.png

4) AO

AO est défini par une Ambient Occlusion map. Cette map est une texture en noir et blanc. Elle contient des ombres typiquement créées par le modèle lui-même. Pour appliquer ces ombres en plus des ombres du modèle de Phong sur le modèle ayant une texture, on multiplie les composantes RGB du texel aux coordonnées (u, v) de la texture, du texel aux coordonnées (u , v) de la AOmap et des ombres calculées à partir du modèle de Phong.

Affichage du modèle WaterBottle.glb avec le programme *m_AO_program* implémentant le fragment shader FragmentAO.glsl

AOmap utilisée : AOmap.png

Affichage du modèle Duck.glb avec le programme *m_AO_program implémentant le fragment shader* FragmentAO.glsl

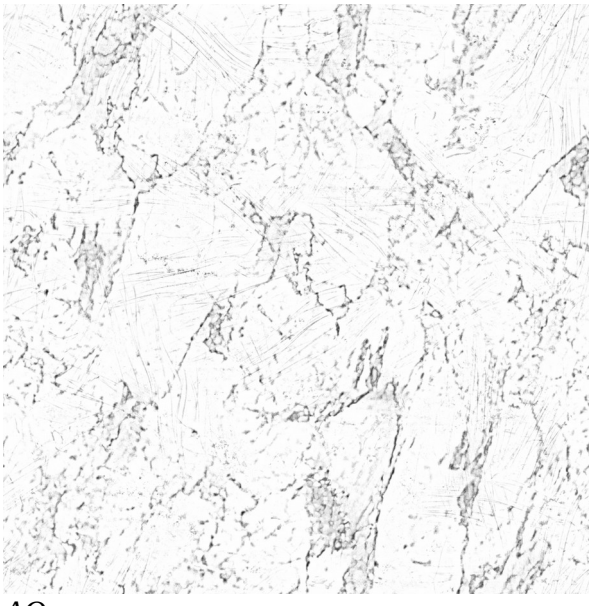
AOmap utilisée : AOmap.png

Affichage du modèle WaterBottle.glb avec le programme *m_AO_program* implémentant le fragment shader FragmentAO.glsl

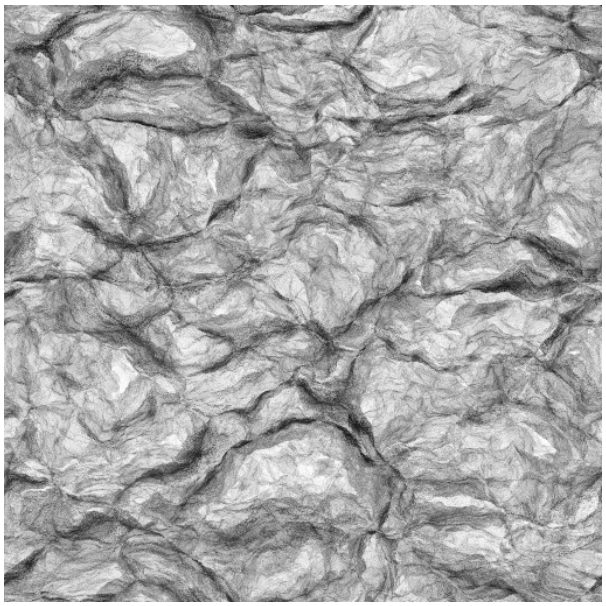
AOmap utilisée : AOmap2.png

Affichage du modèle Duck.glb avec le programme *m_AO_program implémentant le fragment shader* FragmentAO.glsl

AOmap utilisée : AOmap.png



AOmap.png



AOmap2.png

Niveau 1 : Rajouter l’effet d’émission et d’ambiant occlusion.

Niveau 2 : Choisir un modèle PBR connu parmi les suivants et essayer de développer certaines fonctionnalités de ces derniers : glTF PBR, Filament, Dassaut.

Les niveaux 1 et 2 n’ont pas été implémentés.

Sources :
https://cglearn.eu/pub/advanced-computer-graphics/physically-based-shading
https://conceptartempire.com/ambient-occlusion-map/
(pour les définitions dans le TP9 Niveau 0)