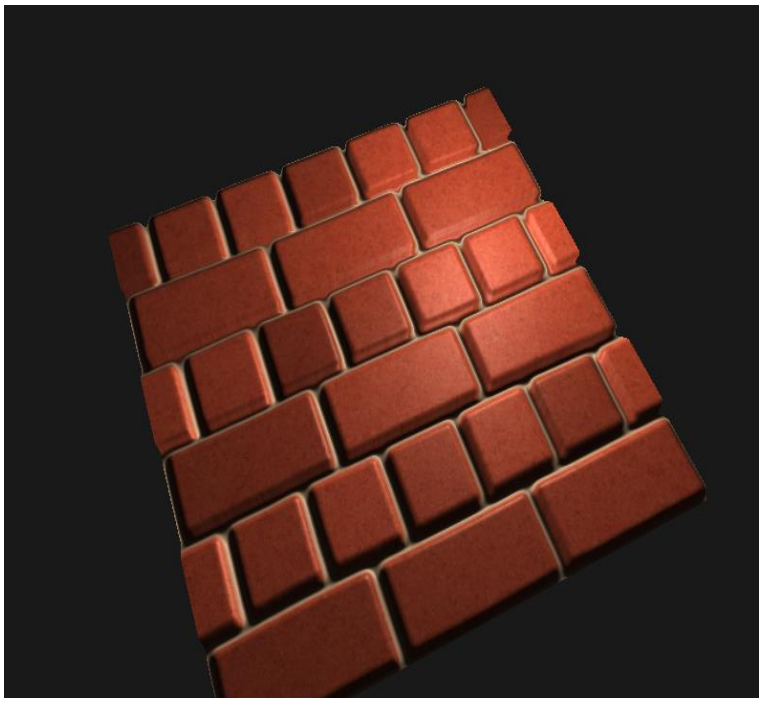
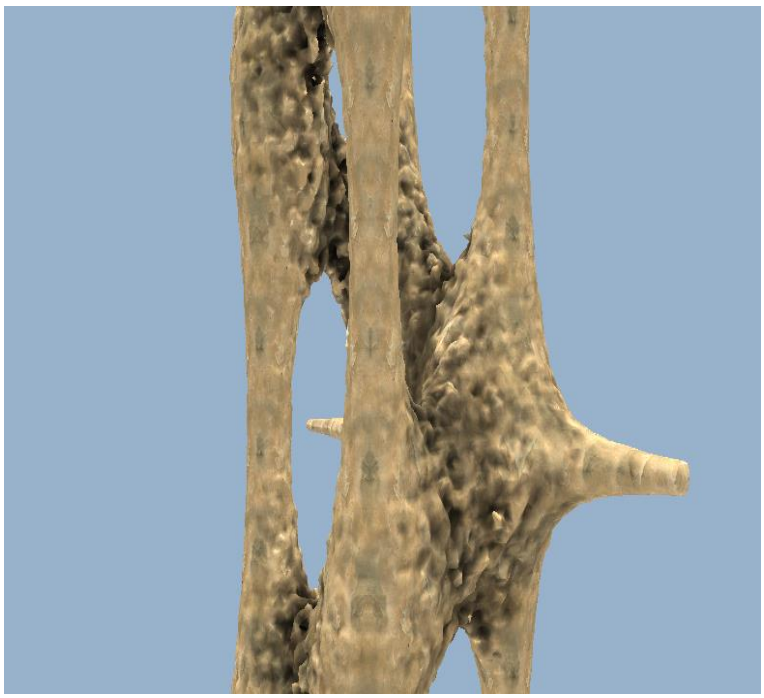
BADOUX Mathieu
LEGAULT Bastien

5ETI

TP Shaders



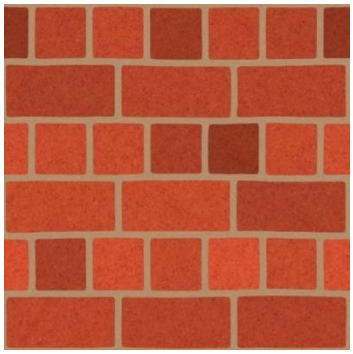


Lors de ce TP, nous avons tout d'abord implémenter l'effet parallax qui permet d'avoir une illusion de relief sur une surface plane. Nous verrons ensuite l'algorithme des marching cubes qui permet de modéliser des surfaces assez complexes tout en leur appliquant une texture.

1- Parallax mapping

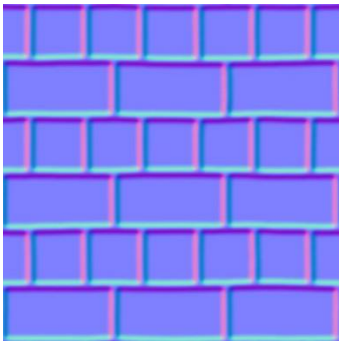
Question 1 :

En plus de l'information « couleur » de la texture, nous avons deux autres textures qui sont appliquées sur notre modèle. La première gère la couleur en chaque point. La deuxième gère la variation des normales en certains points. En effet, grâce à l'illumination de Phong, on sait calculer une illumination en prenant en compte les normales d'un objet 3D. Ici, le relief est simulé avec ce changement de normal en chaque point. En exécutant le programme, nous voyons bien que les tuiles semblent être en relief car une ombre se crée en fonction de la source de lumière.



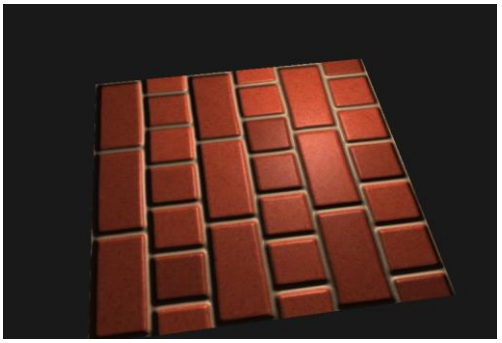
Texture de la couleur

+



Texture des normales

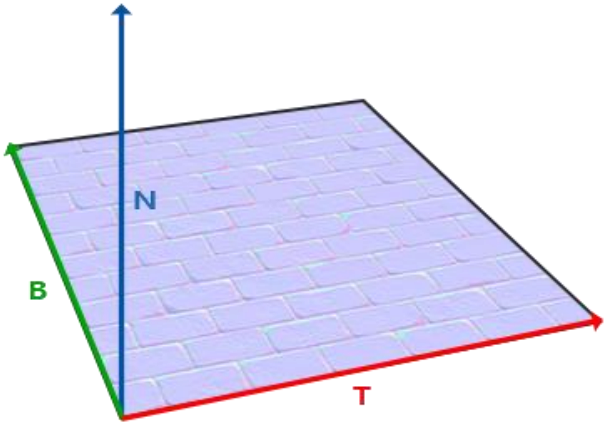
=



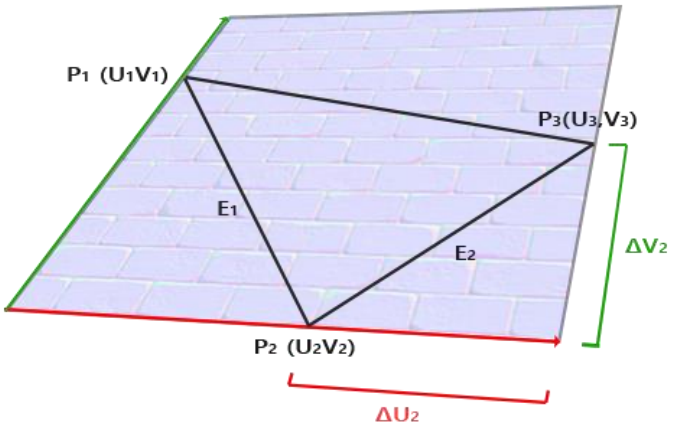
Résultat

Question 2 :

Nous formons les 3 vecteurs tangente, bitangente et normal. Le vecteur normal est le vecteur normal au plan, soit l'axe z.
La direction des vecteur tangente et bitangent suit la direction des axe x et y comme le montre le schéma suivant :



A partir de ce repère, nous pouvons calculer les vecteurs tangente et bitangente liés à chacun des triangles.



Nous avons donc les équations suivantes :

$$(E_{1x},E_{1y},E_{1z})=\Delta U_1(T_x,T_y,T_z)+\Delta V_1(B_x,B_y,B_z)$$
$$(E_{2x},E_{2y},E_{2z})=\Delta U_2(T_x,T_y,T_z)+\Delta V_2(B_x,B_y,B_z)$$

En isolant T et B, nous trouvons :

$$\begin{bmatrix} Tx & Ty & Tz \\ Bx & By & Bz \end{bmatrix} = \frac{1}{\Delta U_1\Delta V_2 - \Delta U_2\Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

Nous retrouvons bien ces équations dans le main.cpp.

Dans ce cas particulier, les 2 triangles sont symétriques et ont un côté en commun, on aurait donc pu éviter les seconds calculs et repartir des premiers résultats en les adaptant au signe près ou en changeant les coordonnées suivant x en suivant y. De plus, certaines coordonnées sont nulles. En effet, les coordonnées choisit pour les deux triangles créent des vecteurs dont certains sont uniquement selon la direction x ou y. Nous aurions donc pu simplifier les premiers calculs.

La fonction glActiveTexture permet de sélectionner une texture courante, elle est associée à la fonction glBindTexture qui associe la texture en paramètre à cette texture courante.

Question 3 :

Pour communiquer entre les deux shaders, nous utilisons différentes variables :
- vf_text_coords reste dans le repère objet car on applique la texture avec cette variable, il faut donc rester dans le repère de l'objet.
- vf_frag_pos passe dans le repère objet grâce à la matrice model ce qui nous aidera pour les calculs de position plus tard.
- les variables vf_tangent_... sont convertis dans le repère objet (tangente, bitangente, normale) grâce à la matrice TBN. On les utilise ensuite pour le calcul de l'illumination.

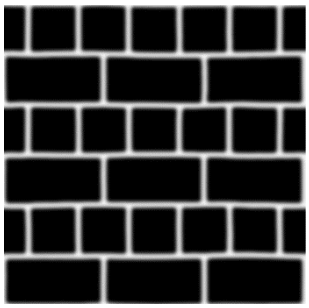
Question 4 :

Pour implémenter l'algorithme de parallax, nous utilisons une height_map. C'est une texture qui référence la profondeur de chaque pixel. Nous récupérons donc la hauteur de chaque pixel via cette texture. Nous calculons ensuite le décalage des coordonnées de texture lié à cette hauteur. On soustrait ensuite aux coordonnées de texture d'origine ce décalage.

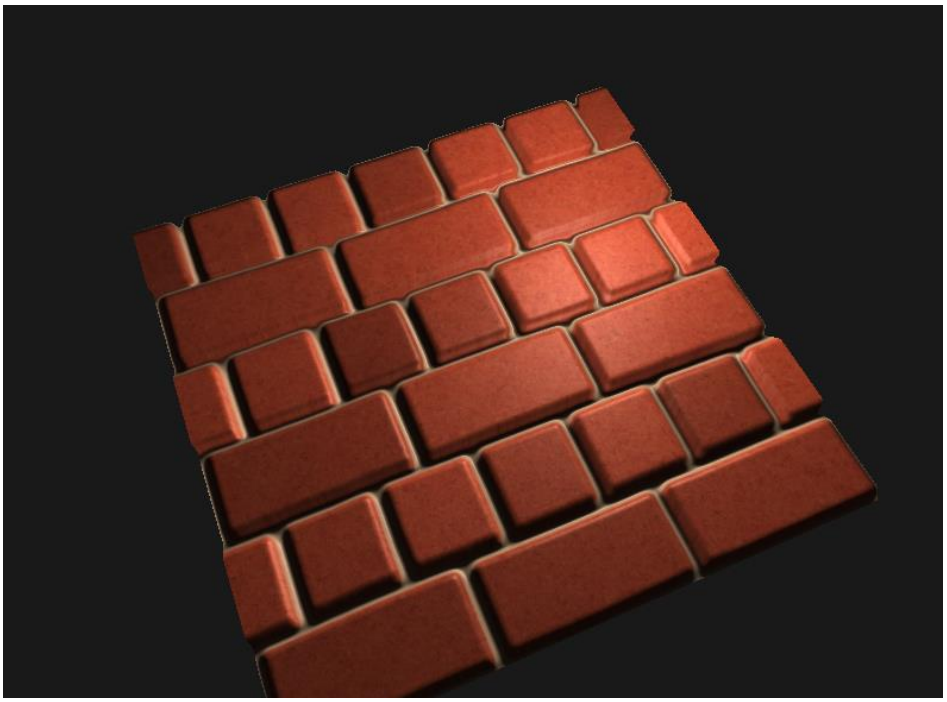
Nous implémentons ce code dans le fragment shader :

```
float height = texture(height_tex, text_coords).r;
vec2 p = viewDir.xy /viewDir.z * (height * height_scale);
text_coords =  text_coords - p;
```

La variable height_scale est envoyé du main. De plus, cette variable est modifiée par l'appui de la touche 'e' ou 'E' pour faire évoluer l'effet de profondeur.



Texture des profondeurs

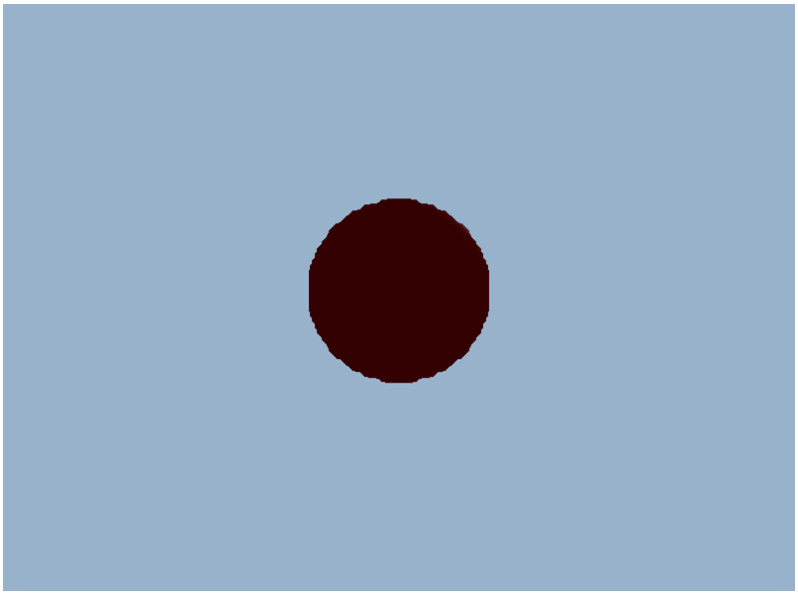


Résultat avec le parallax mapping

2- Marching cubes

Question 5 :

Dans une deuxième partie, nous allons nous intéresser à comprendre et améliorer un algorithme des marching cubes.
Nous récupérons donc un code permettant le calcul et l'affichage de triangles formant une sphère. Nous avons donc comme résultat de départ la figure ci-dessous :



Tout d'abord, nous essayons de comprendre comment fonctionne l'affichage de notre sphère. Notre algorithme est composé en quatre grandes étapes, qui sont les suivantes.

Une première étape consiste en la création d'une fonction de densité pour chaque position 3d. Pour cela, on utilise la fonction « create_build_density_shader » qui va créer un shader build_density et ensuite lui attribuer les positions des vertexs. Cette étape est contenue dans le shader build_density.

La deuxième étape est de créer un shader permettant le listage du nombre de triangles et des intersections correspondantes. Plus précisément, ce shader permet de stocker les numéros des arrêtes qui contiennent les intersections pour chaque cube. Cette étape est contenue dans le shader list_triangles.

Dans une troisième étape, l'algorithme créer un shader effectuant la génération des triangles, le calcul des positions, des normales et de l'occlusion ambiante des sommets. Cette étape est contenue dans le shader gen_vertices.

La dernière étape de l'algorithme est la création du shader effectuant le rendu des triangles. Cette étape est répétée à chaque affichage et elle effectue rendu des triangles de l'étape au-dessus. Cette étape est contenue dans le shader render.

Les données retournées par le shader build_density sont transmise par un FBO dans la fonction compute_density_pass_init.

On utilise des transform feedback pour passer les informations des triangles du shader list_triangle (outtripos et outtriedge) ainsi que la position et les normales du shader gen_vertices (outvert et outnorm).

Question 6 :

La fonction glDrawArraysInstanced permet de dessiner plusieurs instances d'une gamme d'éléments. Dans cette fonction on précise quel type de primitives on veut dessiner, l'indice de départ, le nombre d'indice à dessiner et le nombre d'instances à dessiner.
Dans notre code, on aurait pu se passer de la fonction mais il aurait fallu compter de nous-mêmes le nombre d'instance dessiner et utiliser la fonction glDrawArrays dans une boucle for.

Question 7 :

La fonction glActiveTexture permet de sélectionner quelle texture il faut activer.
Elle prend deux paramètres, le premier est l'échantillonneur qui est liée aux texels que l'on extrait avec la texture, le deuxième est les coordonnées de texture à laquelle la texture va être échantillonnée.

Question 8 :

Dans les shaders, on peut trouver des variables du type sampler2D et isampler2D. La différence entre les deux est que lorsque l'on met un i devant par exemple ivec4, on précise que le vec4 créer sera composé d'entier tandis qu'un vec4 sera simplement un vecteur qui possède 4 composantes flottantes.

La fonction TexelFetch sert à rechercher un unique texel dans une texture.

Question 9 :

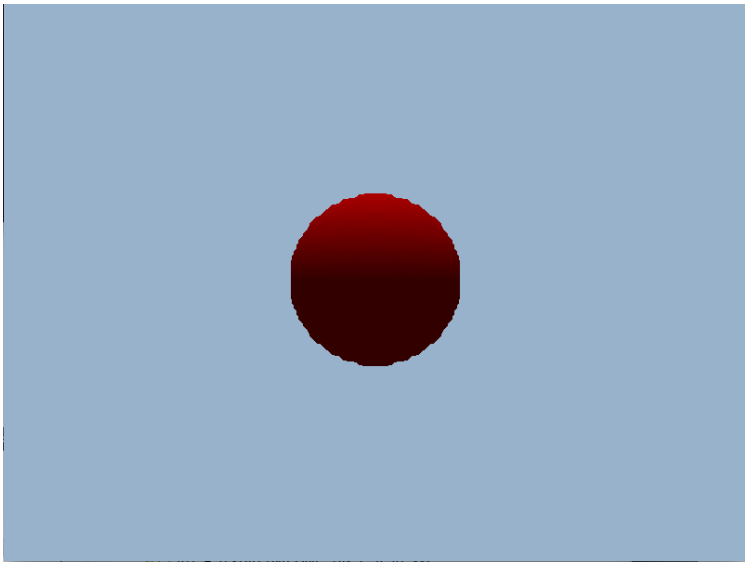
Pour avoir un meilleur affichage de notre sphère, nous avons dû modifier plusieurs paramètres.
Tout d'abord, nous avons dû modifier le calcul des normales de nos sphères. Pour cela, nous avons utilisé le gradient de la fonction de densité. Nous avons calculé les normales dans notre vertex shader gen_vertices.

Pour calculer les normales selon les 3 axes, nous avons utilisé le calcul suivant qui pour chaque axe, fait la différence entre la densité de texture de notre point plus un offset et la densité de texture de notre point moins un offset.

On utilise le code suivant pour faire le calcul :

```
vec3 grad = vec3(1.);
float d = 1.0/256.0;
 grad.x = text_density(pos + vec3( d, 0, 0)) -
      text_density(pos + vec3(-d, 0, 0));
 grad.y = text_density(pos + vec3( 0, d, 0)) -
      text_density(pos + vec3( 0,-d, 0));
 grad.z = text_density(pos + vec3( 0, 0, d)) -
      text_density(pos + vec3( 0, 0,-d));
 grad = normalize(grad);
```

Une fois le calcul effectué pour les 3 axes, nous normalisons notre vecteur de normal.
On obtient le résultat suivant en effectuant seulement le calcul des normales :



Nous obtenons donc une meilleure représentation de notre sphère mais le résultat n'est toujours pas celui que l'on souhaite. Nous avons toujours un effet cranté sur le contour de notre sphère.

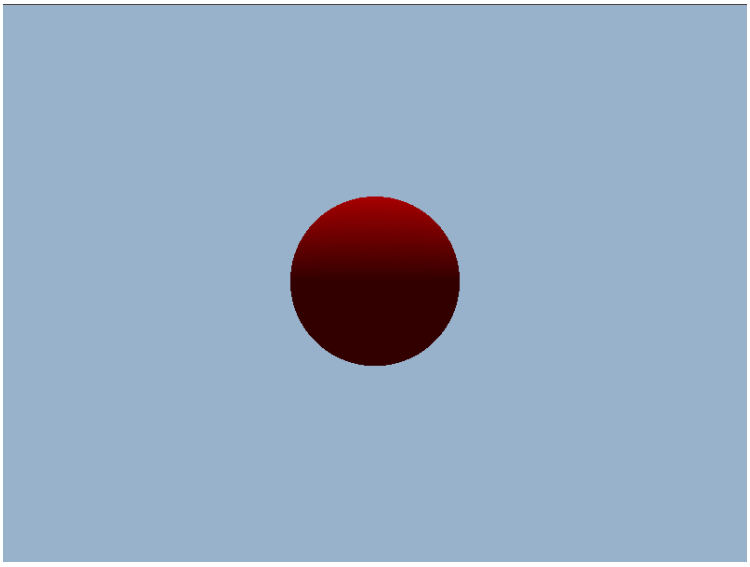
Question 10 :

Nous modifions alors l'interpolation de nos points pour obtenir une sphère lisse. Pour l'instant, nous positionnons nos points au milieu des arêtes de nos cubes. Nous allons donc mettre en place une interpolation linéaire pour mieux positionner nos points sur ces arêtes. Nous utilisons la formule suivante pour calculer ces positions :

$$\text{Interpolation} = v_0 + (v_1 - v_0) * \frac{l_0}{(l_0 - l_1)}$$

Avec v0 et v1 les positions des angles du cubes
Avec l0 et l1, la distance respective entre notre point et les angles v0 et v1.

Nous obtenons une sphère bien lisse comme résultat après interpolation linéaire comme on peut le voir sur la figure ci-dessous :



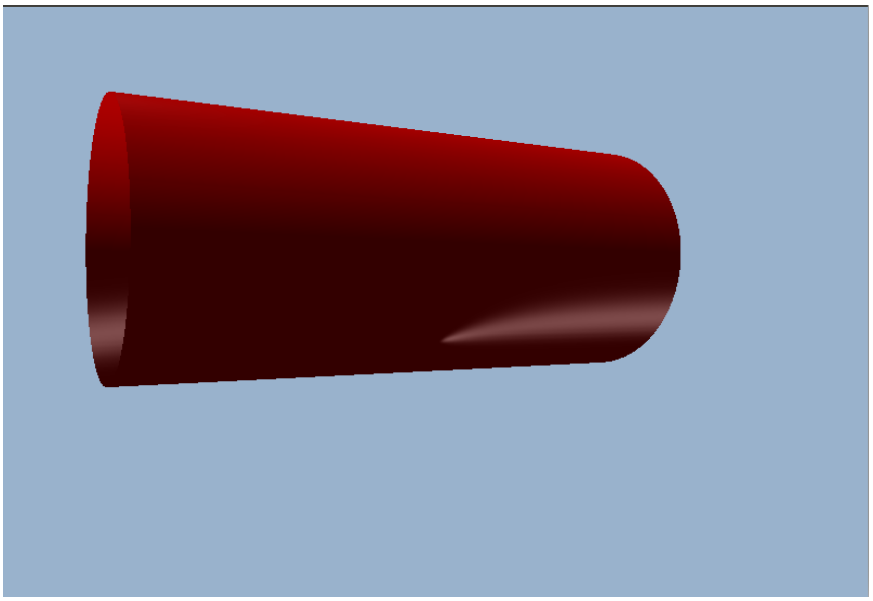
Question 11 :

Nous avons pour l'instant la fonction de densité d'une sphère qui est égale à l'équation suivante :

$$densité = 1.0 - length(ws)$$

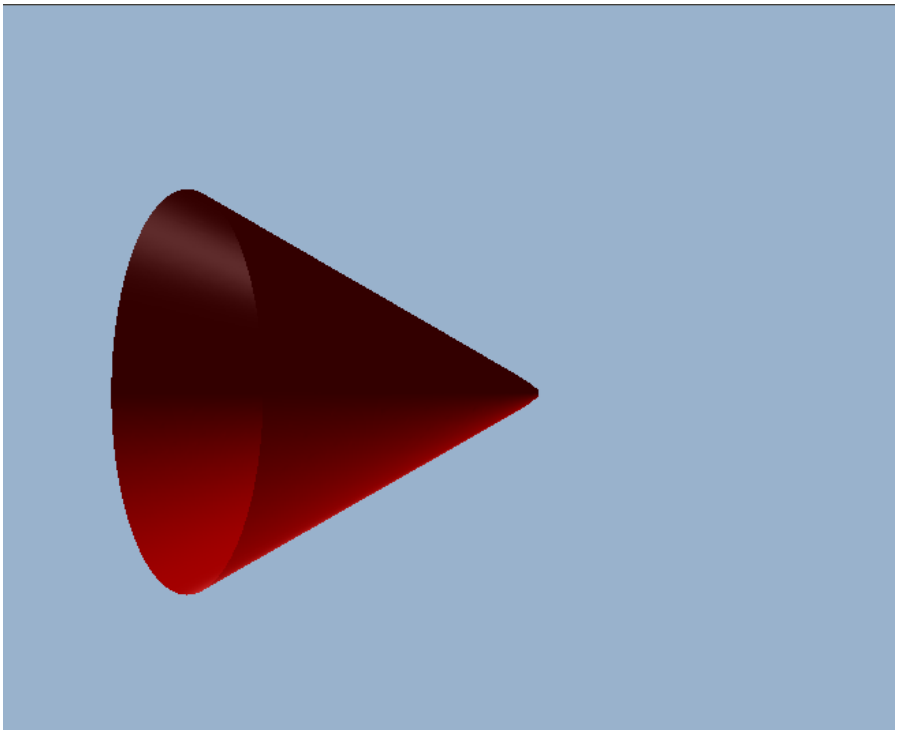
En modifiant cette fonction de densité dans le fragment shader de build_density, nous pouvons obtenir différentes formes telle qu'un cylindre avec la formule suivante :

$$densité = 1.5 - length(ws.xz - vec2(1.0, 1.0))$$



On peut également obtenir un cône avec la formule de densité suivante :

$$densité = dot(vec2(0.7,0.3), vec2(length(ws.xz), ws.y))$$



Dans notre cas, on cherche à construire une forme plus complexe qu'une sphère, un cylindre ou un cône.

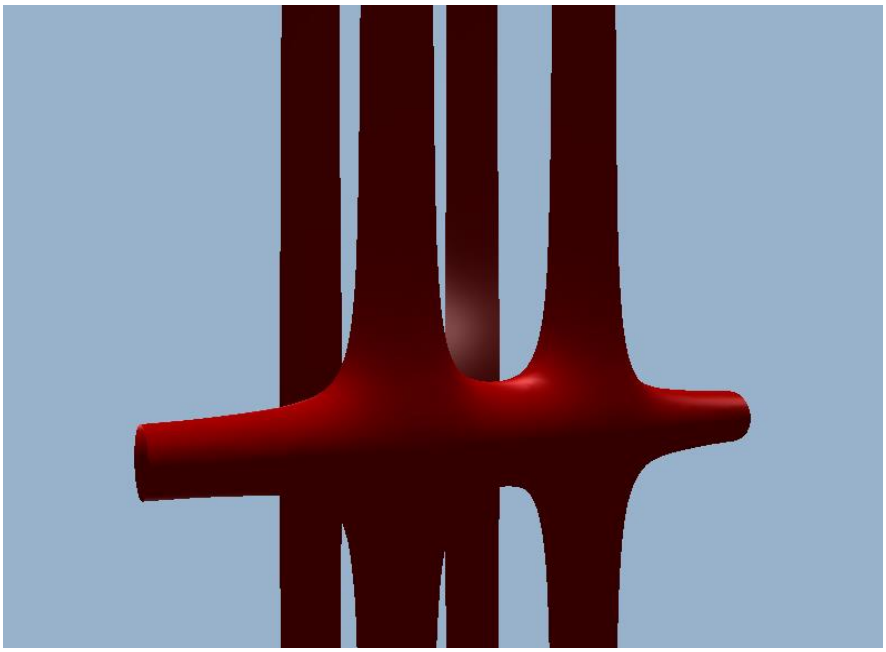
Pour cela, dans notre densité de fonction, nous allons ajouter les formes que l'on veut obtenir.

On commence par ajouter 4 cylindres verticaux en 4 positions avec le code suivant :

```
f += 1.0 / length(ws.xy - vec2(1.0,1.0)) - 1.0;
f += 1.0 / length(ws.xy - vec2(-1.0,1.0)) - 1.0;
f += 1.0 / length(ws.xy - vec2(-1.0,-1.0)) - 1.0;
f += 1.0 / length(ws.xy - vec2(1.0,-1.0)) - 1.0;
```

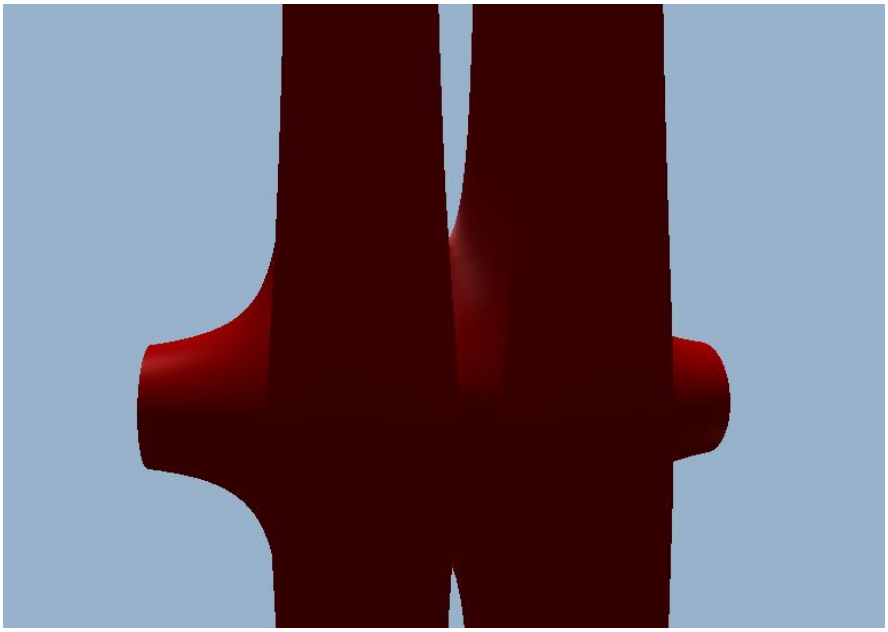
On peut ensuite ajouter un cylindre horizontal avec la ligne de commande suivante :

```
f += 1.0 / length(ws.xz - vec2(1.0,-1.0)) - 1.0;
```



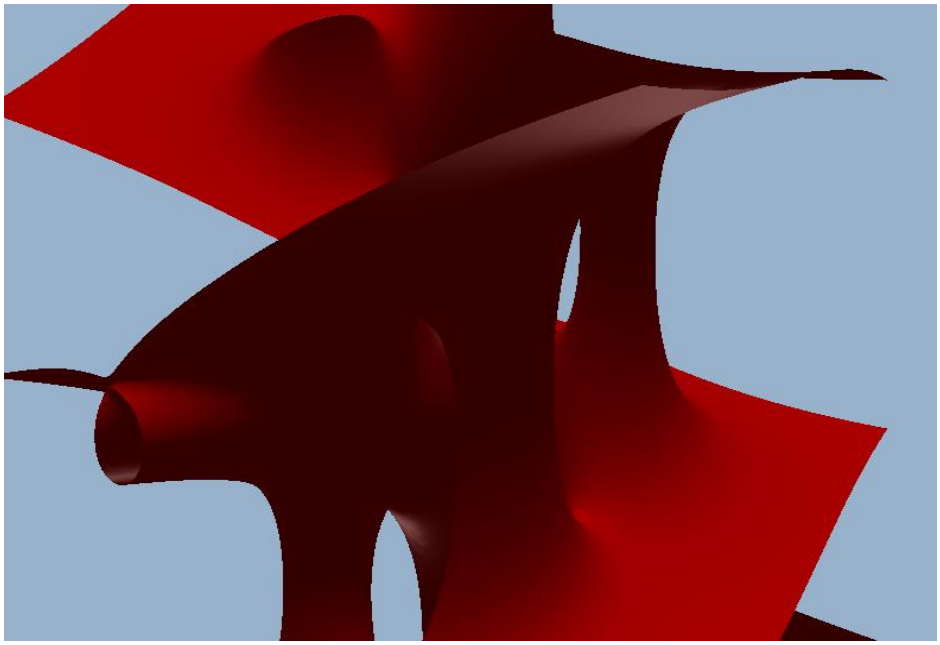
On peut ajouter un pilier inversé au centre de notre structure. Cela équivaut à créer un pilier de vide au centre de notre image.

```
f -= 1.0 / length(ws.xy) - 2.5;
```



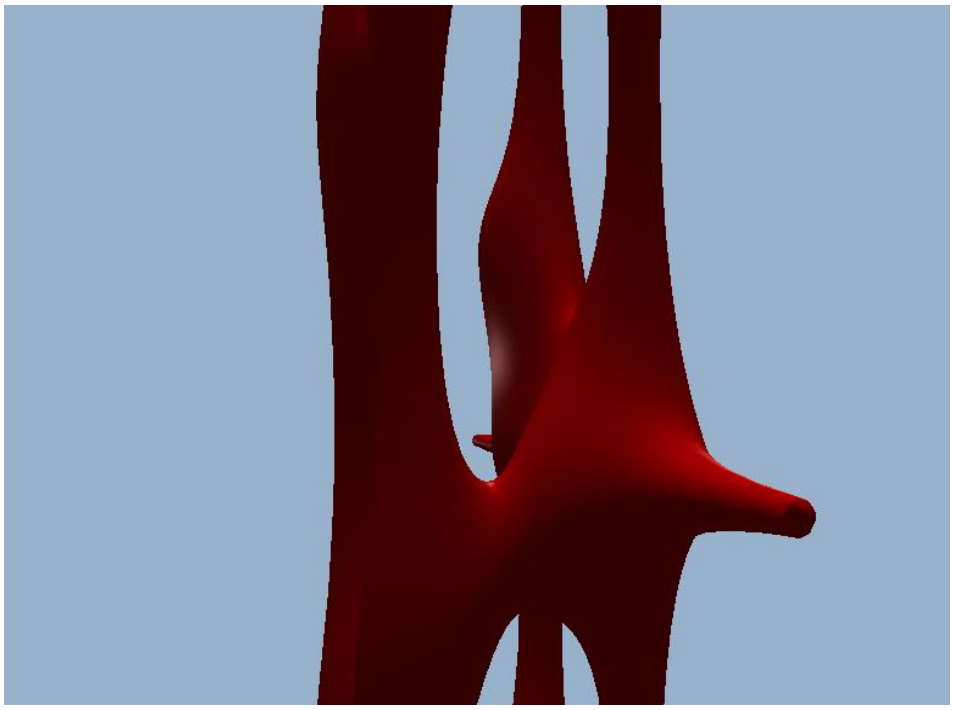
On rajoute également une hélice qui donne le résultat suivant :

```
vec2 vec = vec2(cos(ws.z), sin(ws.z));
f += dot(vec, ws.xy);
```



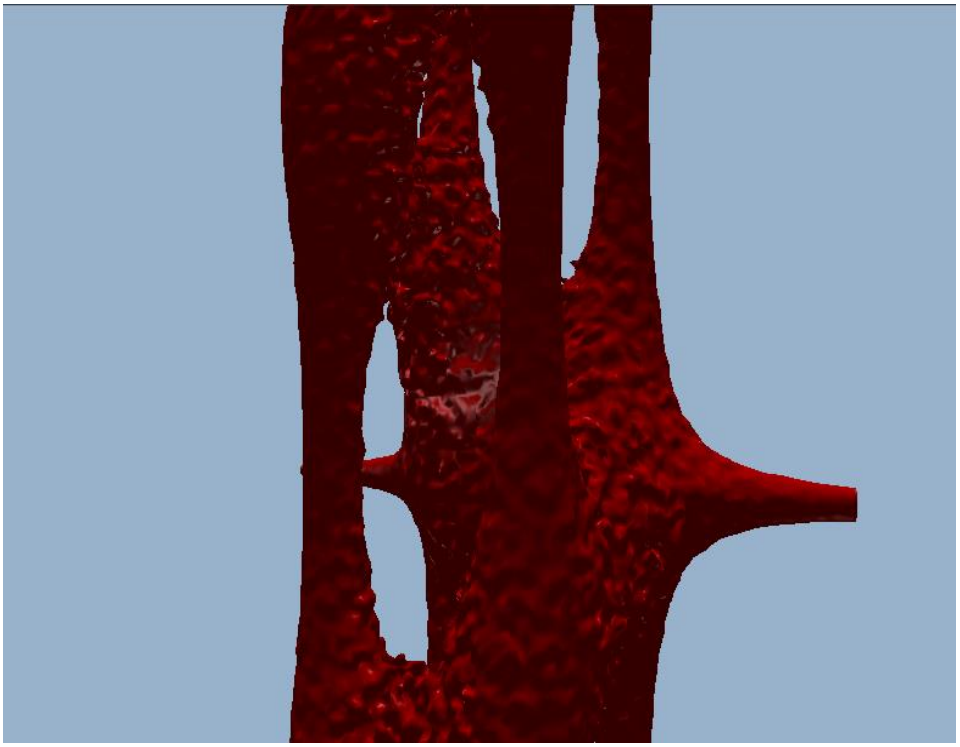
On obtient le résultat suivant après ajout du dernier élément qui ajoute de fortes valeurs négatives sur les bords extérieur :

```
f -= pow(length(ws.xy), 2);
```



On peut ensuite ajouter du bruit à notre structure afin d'obtenir un meilleur rendu, on obtient le résultat suivant :

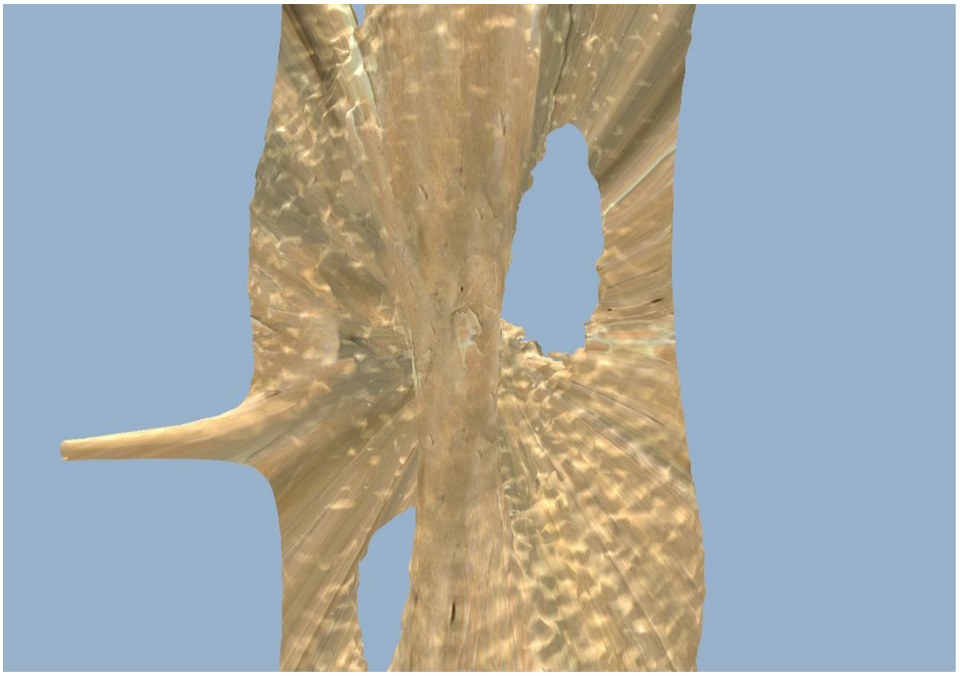
```
color += Noise_MQ_unsigned(ws, noiseVol0);
```



Question 12 :

On cherche maintenant à appliquer une texture sur notre image.

Une première manière de « mal » appliquée la texture est de l'appliqué simplement en x et y sans se préoccuper de l'axe z. On obtient le résultat suivant pour cette application de texture :

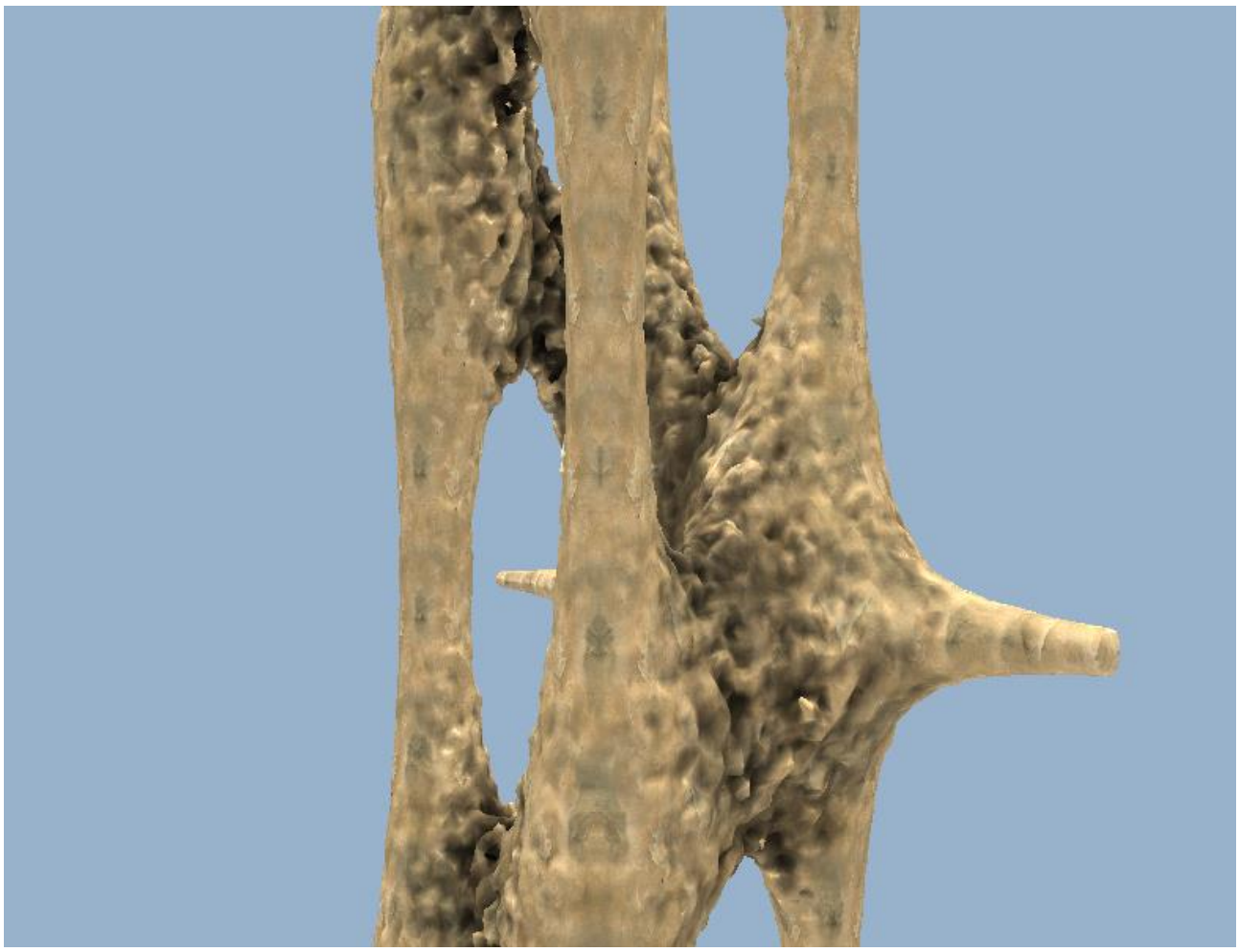


Pour appliquer la texture sur notre objet, on définit des poids selon x, y et z que l'on calcule comme étant le rapport de la normale selon x, y ou z avec la somme des normales en x, y et z.
On applique ensuite la texture transverse à notre axe en appliquant chacun des poids.
Pour cela, on applique le code suivant :

```
vec4 blended_color;
float sum_n = abs(n.x)+abs(n.y)+abs(n.z);
float poidsx = abs(n.x)/sum_n;
float poidsy = abs(n.y)/sum_n;
float poidsz = abs(n.z)/sum_n;
blended_color = (poidsx * texture(myTexture,vf_position.yz) +
            poidsy * texture(myTexture,vf_position.xz) +
            poidsz * texture(myTexture,vf_position.xy)
             );
```

Avec cette ligne de code, nous obtenons le résultat suivant en appliquant une texture.

```
color = occ_light * occ_light * (blended_color * 0.8 + 0.6 * diffuse + 0.3 * vec4(vec3(specular), 1.0));
```



Nous avons donc une meilleure application de la texture sur notre objet qu'en appliquant la texture simplement en x et en y.