

# Compte Rendu - TP ETI5-IMI

## *Shaders avancés et Marching Cubes*

Di Folco Maxime - Girot Charly

27/10/2017

## 1 Parallax Mapping - Donner l'illusion du relief

### 1.1 Utilisation des textures de normales

Habituellement, nous utilisons des textures *colorées* appliquées directement sur un maillage. Exemple : dessiner un mur de briques comme sur la Fig.1

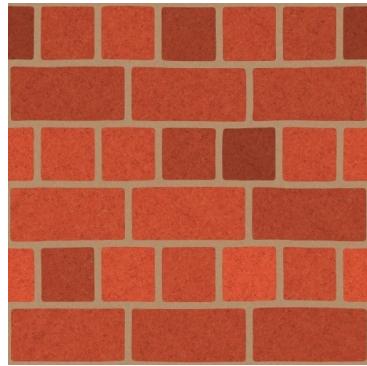


FIGURE 1 – Texture colorée de briques

Dans ce TP, nous souhaitons utiliser des textures contenant d'autres informations afin de les utiliser pour modifier des caractéristiques de nos images 3D comme les normales sur la Fig.2(a), ou pour simuler un effet de profondeur comme sur la Fig.2(b) dans le but d'avoir un rendu plus réaliste.

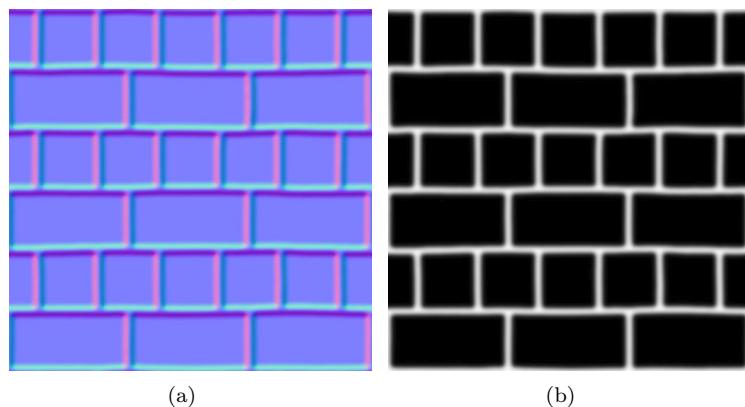


FIGURE 2 – Textures contenant les informations de normales (a) et de profondeur (b)

Dans un premier temps, on nous fournit un programme qui implémente le normal mapping, dont le résultat est présenté Fig.3(b). Cette méthode permet de simuler graphiquement des détails géométriques sur les surfaces représentées. En effet, dans cette méthode, les normales ne sont plus seulement orientées selon l'axe principal  $z$  de la surface mais sont désormais liées aux axes  $x, y$  et  $z$  de la surface en fonction de la carte des normales représentée Fig.?? où la composante rouge indique une orientation de la normale modifiée selon l'axe  $x$ , verte pour l'axe  $y$  et bleue pour l'axe  $z$ . Les normales ne sont ainsi plus toutes orientées dans le même sens selon la surface de l'objet, mais orientées différemment selon chaque fragment constituant notre objet comme schématisé Fig.1.1. On obtient ainsi grâce aux nouvelles orientations de normales, des surfaces contenant plus de détails et donc une impression de relief et un réalisme supérieure comparée à la Fig.3(a).



FIGURE 3 – Résultat de l’application de texture couleur (a) auquel on applique la gestion des normales (b)

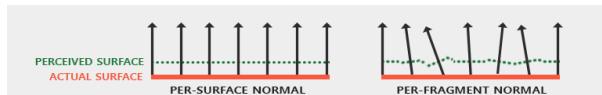


FIGURE 4 – Principe de l’orientation des normales en parallax mapping - tiré de : <https://learnopengl.com/#!Advanced-Lighting/Normal-Mapping>

Lors de l’application d’un normal mapping, lorsque la carte des normales possèdent majoritairement des normales selon un axe, il faut que la normale interpolée de la surface soit de même direction. Si tel n’est pas le cas, on se retrouve alors avec un problème d’illumination comme le montre la Fig.1.1. En effet les normales ont été calculées pour chaque fragment selon la carte des textures et non pas selon l’orientation de la normale surfacique.

Pour résoudre ce problème d’illumination selon les normales, on se place dans l’espace TBN (tangentes, bitangentes, Normales) de chaque triangle. Dans cet espace, les normales y sont calculées dans ce qui se trouve être un espace local pour chaque triangle. Les normales calculées et modifiées par la carte des hauteurs pointent alors toutes approximativement dans la direction  $z$  de chaque triangle peu importe l’orientation finale de l’objet. Il est alors possible de transformer les normales depuis l’espace local tangent vers l’orientation finale de la surface.

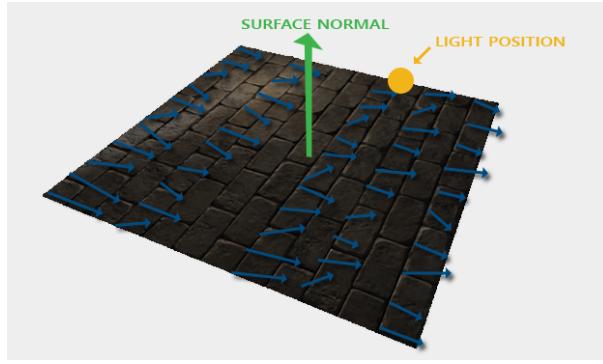


FIGURE 5 – Problème d’orientation des normales pour l’éclairage lors d’un normal mapping classique - tiré de : <https://learnopengl.com/#!Advanced-Lighting/Normal-Mapping>

Pour pouvoir utiliser la méthode de normal mapping, puis ensuite la méthode de parallax que nous allons détailler par la suite, il faut utiliser de nombreuses variables pour pouvoir calculer la matrice TBN et ensuite calculer les directions de la vue et de la lumière dans nos shaders. Dans le Vertex Shader nous avons besoin des :

- sommets habituels : `in vec3 position`
- données des normales : `in vec3 normal`
- coordonnées de textures : `in vec2 tex_coords`
- tangeantes et bitangeantes calculés précédemment : `in vec3 tangent ; in vec3 bitangent`

On calcule alors la matrice TBN pour passer de l'espace local tangent vers l'orientation finale de la surface, avec les lignes de codes suivantes :

```
out vec3 vf_frag_pos = model * position;
out vec2 vf_tex_coords;
out vec3 vf_tangent_light_pos = TBN * light_pos;
out vec3 vf_tangent_view_pos = TBN * view_pos;
out vec3 vf_tangent_frag_pos = TBN * vf_frag_pos;
```

## 1.2 Ajout d'un effet de profondeur avec l'utilisation des cartes de hauteur

Le normal mapping nous a permis d'obtenir un niveau de détails supplémentaire sur nos textures. Avec peu de données supplémentaires, il est possible de donner une véritable impression de relief en réalisant un Parallax Mapping. Pour cela nous allons utiliser une carte de hauteur afin de modifier par projection les coordonnées de textures à afficher. On utilisera donc dans les shaders une variable supplémentaire `height_tex` afin de connaître l’élévation correspondante à chaque coordonnées de textures. Cette méthode simule une impression de relief.

Supposons que nous sommes à une position `view_pos` et regardons vers la position `frag_pos` comme indiqué Fig.1.2. Le point réel observé ne devrait pas être `frag_pos` correspondant à la coordonnée de texture de départ mais le point C. Il faut alors calculer le point C par projection, ce qui s'avérerait trop complexe. Pour simplifier le problème on cherche alors la position du point P par calcul de l'offset entre le point A (`frag_pos`) et la projection approximative du point B sur la surface en utilisant la hauteur au point A. Le résultat de cette méthode est présenté Fig.7(b).

**Pourquoi ça marche mieux quand on a une vue rasante. on applique ensuite l'algorithme suivant pour ajouter l'effet de profondeur**

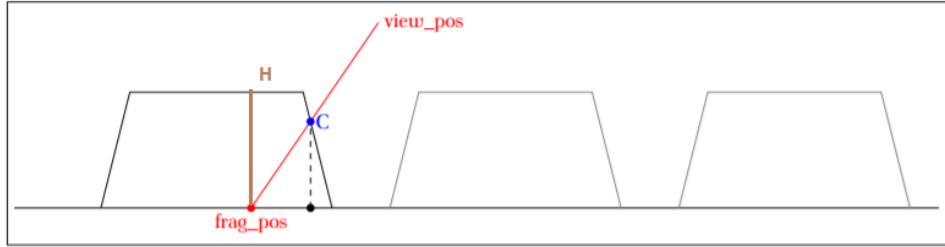


FIGURE 6 – aaa



FIGURE 7 – Comparaison de la gestion des normales (a) auquel on applique une carte de hauteur (b)

Néanmoins, lors du calcul de nos hauteurs nous utilisons un facteur qui permet d'augmenter ou diminuer les hauteurs des coordonnées de nos textures. Si ce facteur est trop petit ou trop grand le rendu ne paraît plus réaliste. En effet, l'offset entre les coordonnées de texture réels et les modifiées devient trop important. La surface supérieure des briques semble alors se séparer du mur, ce qui provoque une perte complète du réalisme de la scène.

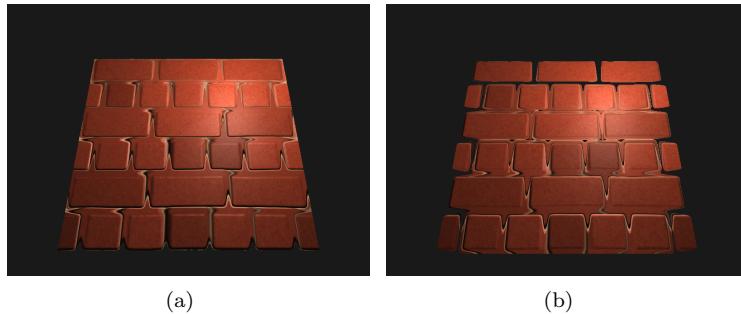


FIGURE 8 – Résultats de la carte des profondeurs pour un facteur de hauteur trop faible (a) trop haut (b)

## 2 Marching Cubes

### 2.1 Présentation Générale

L'algorithme des *Marching Cubes* a pour but d'extraire des meshes à partir d'un ensemble de points de l'espace, défini comme un champ scalaire où à chacun des points 3D est associé sa densité. Une valeur positive de densité correspond à un point de la surface solide et une valeur négative à un espace

vide. Grâce au GPU, l'espace est subdiviser en  $3*3*8$  blocs de  $48*48*48$  blocs élémentaires (voxels) reprezentés par des cubes. C'est au sein de chacun des cubes que l'algorithme remplace et il faut alors distinguer  $256$  ( $2^8$  : 8 angles pour le cube) cas différents :

- Cas 0 : Les 8 angles du cube sont de densité négative : le cube est complètement en dehors de la surface => Rien à dessiner.
- Cas 1 → 254 : Au moins un des 8 angles n'est pas de même signe que les autres : On cherche alors au sein d'une table de correspondance pour savoir combien de polygones seront générées au sein du cube et comment les construire. La Fig.2.1 présente les principaux cas possibles parmi les 256 possibles.
- Cas 255 : Les 8 angles du cube sont de densité positive : le cube est completamente à l'intérieur de la surface => Rien à dessiner puisqu'on ne represente graphiquement que les bordures d'une surface.

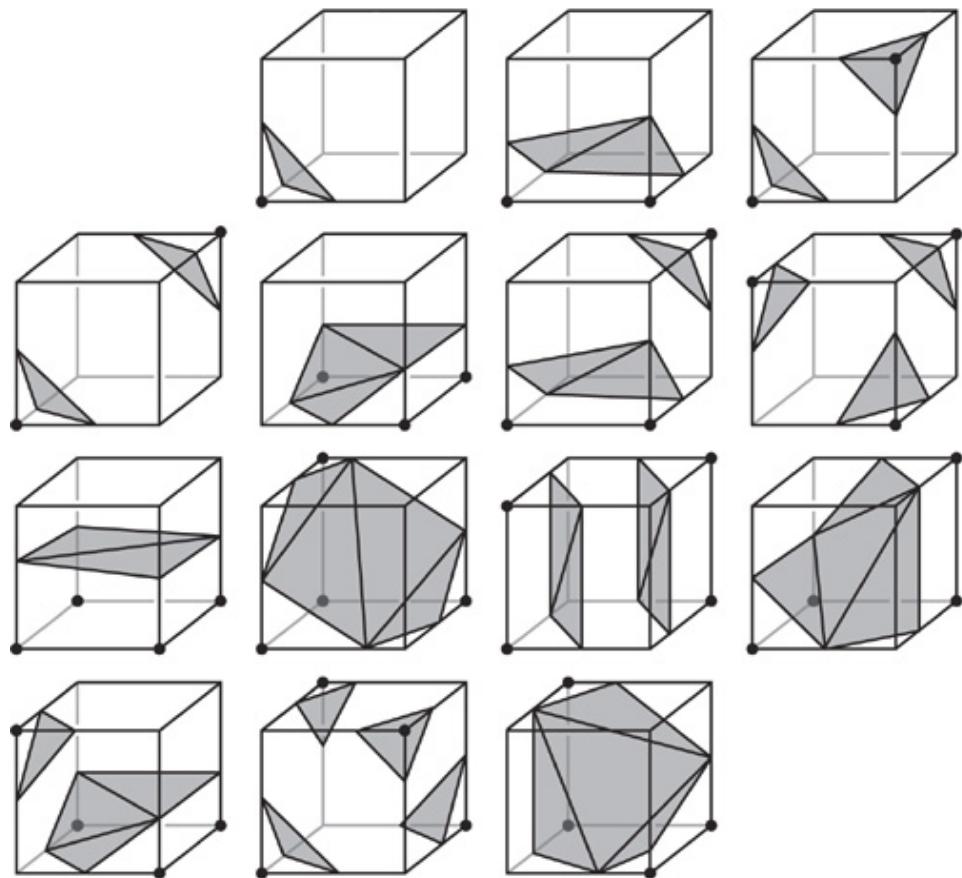


FIGURE 9 – Illustration des principaux cas possibles pour la representation des polygones avec l'algorithme des marchings cubes

La Fig.2.1 montre l'enchainement des différents shaders pour la réalisation des marching cubes. *Ca serait cool si on pouvait juste dire les quels échangent via des FBO et lesquels via des TF (je pense que c'est ça qu'on comprenait pas l'autre jour)*

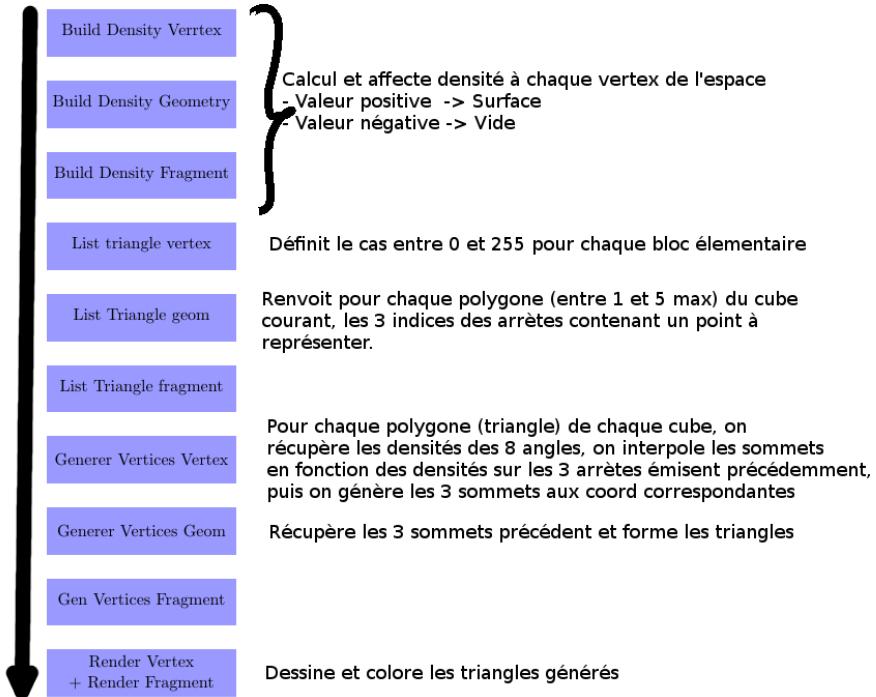


FIGURE 10 – Pipeline pour la réalisation de l'algorithme marching cubes avec explications des blocs principaux

## 2.2 Analyse Du code

Question 6 : Lisez la documentation de la fonction `glDrawArraysInstanced` , que réalise cette fonction ? Aurait-on pu s'en passer ? Quelle est l'utilisation plus classique de cette fonction ?

De la même manière que `glDrawArrays` permet de "dessiner/rendre" une primitive (un triangle dans notre cas), `glDrawArraysInstances` permet de synthétiser une série instanciée de primitives. On aurait donc pu s'en passer en utilisant `glDrawArrays` dans une boucle avec itération des indices de chaque primitive (ce qui est fait automatiquement avec cette fonction).

Question 7 : Comment s'utilise la fonction `glActiveTexture` ?

## 2.3 Amélioration de l'algorithme

Question 8 : Dans les shaders, on trouve des variables de type `isampler2D` et d'autres de type `sampler2D`. Quelle est la différence entre ces deux types ? Que réalise la fonction `texelFetch` ?

Les documentations font références à `gsampler2d` ou `g` est remplacé par rien, `u` ou `i`. Rien signifie que le `sampler2D` contenant des coordonnées de textures sera exprimé en `float`, `i` par des entiers signés et `u` par des entiers non signés.

`TexelFetch` recherche le texel (pixel de texture) correspondant à la coordonnées de texture qui lui a été fournit et à la texture.

Question 9 : Corrigez le calcul des normales en supposant que sa direction est donnée par le gradient de la fonction de densité. Quel fichier avez-vous modifié pour cela ?

Jusqu'à présent le calcul de normales pour chaque triangle était constant (chaque polygone possède la même normale). On choisit alors d'interpoler les normales pour chaque triangle en fonction du gradient

de la fonction de densité en chaque point, le résultat est présenté Fig.11(b).

```
// pos : position du voxel courant
vec3 calc_normal(vec3 pos)
{
    vec3 grad = vec3(1.); // Ancienne normale
    float d = 1.0/48.0; // Ecart entre deux fragments pour calcul du gradient
    // Calcul du gradient dans les 3 directions x,y,z
    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));
    return normalize(grad);
}
```

On cherche encore à améliorer le résultat en interpolant correctement les sommets des points sur chaque arrête. En effet ils étaient précédemment toujours calculés au milieu des arrêtes du cube sans tenir compte de la densité de chaque sommet, ce qui à pour effet de rendre une sphère avec un aspect crénelé. On choisit alors d'interpoler linéairement et le résultat est présenté Fig.11(c) :

```
// v0, v1 : positions des sommets
// l0, l1 : densité des sommets v0 et v1
vec3 vertexInterp(vec3 v0, float l0, vec3 v1, float l1)
{
    // return (v0 + v1) / 2; // Ancienne interpolation
    return (l1*v0 - l0*v1)/(l1-l0); // Interpolation linéaire
}
```

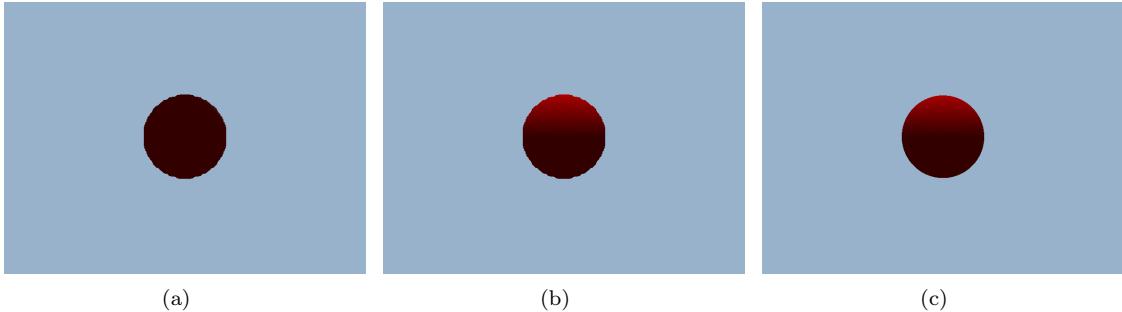


FIGURE 11 – Marching Cubes Original (a) ; Calcul des normales fonction du gradient de la fonction de densité (b) ; Interpolation linéaire de placement des vertex par rapport aux arrêtes (c)

## 2.4 Fonction de densité

Avec le nouveau calcul des normales et les interpolations des sommets corrects, il est désormais intéressant de générer des structures plus chaotique, plus aléatoire, plus intéressantes. On s'inspire alors de l'algorithme de calcul de la densité présenté par NVidia : "Cascades par Ryan Geiss et Michael Thompson". On présente nos différents résultat de fonctions de densité tout d'abord pour des fonctions de densité simple (Sphère, Torus Fig.12(a) et 12(b)) puis par utilisation de l'algorithme NVidia Fig12(c). On fais également varier certains paramètres de cet algorithme (comme suit) pour obtenir notre résultat final Fig.13(b) avec ou sans bruit.

```
/** Return a density function to create a self-shaped rock */
float density_perso(vec3 ws)
{
```

```

//ws coordinates UV of fragments
float f = 0; // density function

//Add noise to fragments so the result isn't flat but irregular
ws.yz += 0.5*Noise_MQ_unsigned(ws/10,noiseVol0).xy
+ 0.5*Noise_MQ_unsigned(ws/10,noiseVol1).xy
+ 0.5*Noise_MQ_unsigned(ws/10,noiseVol2).xy
+ 0.3*Noise_MQ_unsigned(ws/10,noiseVol3).xy;

//Create 4 pillar center in a xy plane (NB : xy OpenGL = xz in most of the case)
vec2 pillar [4];
pillar [0] = vec2(0.4,-0.8);
pillar [1] = vec2(0.4,1.0);
pillar [2] = vec2(-0.8,-0.4);
pillar [3] = vec2(-0.4,1.0);

for (int k=0; k<4; k++)
{
    f += 1 / length(ws.xy - pillar [k].xy) -1; // add positive value at pillar
}
f -= 1 / length(ws.xy) - 3; //Add negative values going down the center (water flow
f = f - 2*pow(length(ws.xy), 3); // Add strong negative values at outer edge (Help w

//Rotate the values as the slice's Z coord changes.
vec2 v = vec2(cos(ws.z),3*sin(ws.z));
f += dot(v,ws.xy);

//Shelves : periodically add positive values based on slice's Z coord.
f += 5*cos(ws.z);

return f;
}

```

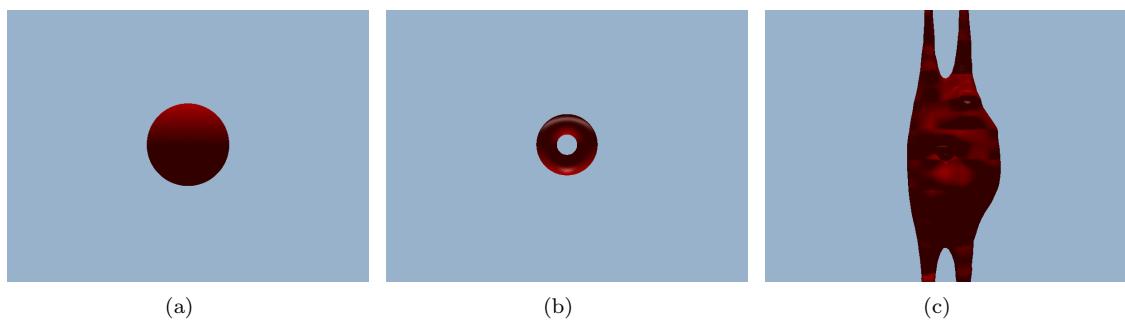


FIGURE 12 – Marching Cubes avec différentes fonctions de densité : Boule (a), Torus (b), Crédit de roches (c)

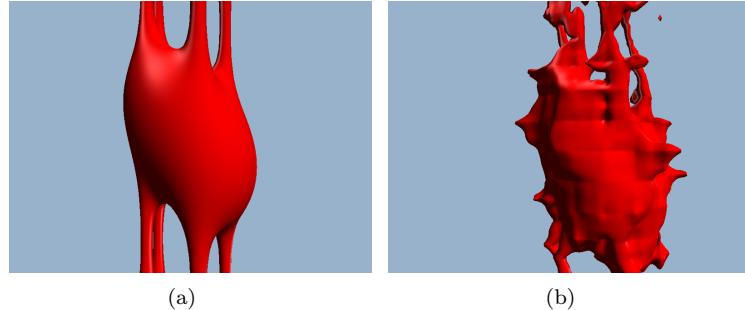


FIGURE 13 – Densité finale retenue non bruitée (a), bruitée (b)

## 2.5 Amélioration graphique & Textures

Le but de cette partie est d'appliquer une texture quelconque à la surface. Or les coordonnées de textures n'existe pas et il serait impossible d'en obtenir des cohérentes au préalable étant donné la surface quelconque. Ce sont ainsi les coordonnées spatiales des sommets qui sont utilisées comme coordonnées de textures en ne prenant que deux dimensions sur les 3. Pour ne pas privilégier une seule direction sur laquelle projette la texture (choisir x et y comme coordonnées de texture reviendrait à projeter la texture selon l'axe z), les deux coordonnées sont choisies par pondération des normales. Les textures sont alors projetés sur l'axe prépondérant et les 2 autres axes sont utilisés comme coordonnées de texture. La Fig.14 représente cette affichage avec des couleurs pondérées par les normales : Rouge(x), Vert(y), Bleu(z).

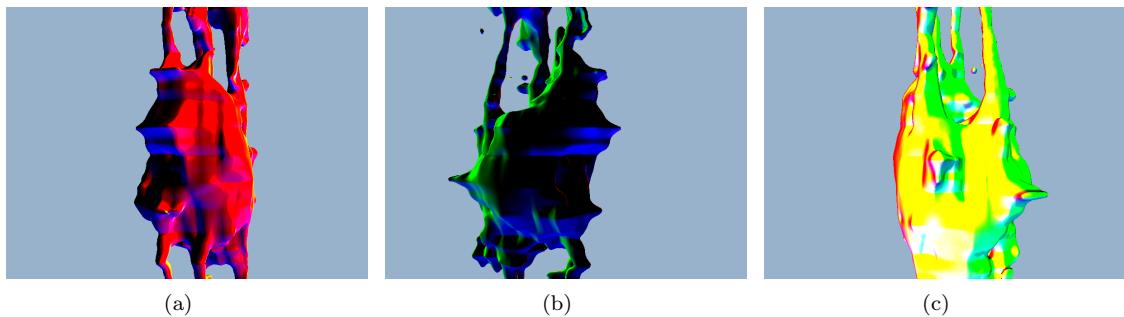


FIGURE 14 – Application des normales pondérés représentées par le mélange des 3

Enfin, on remplace l'affichage couleurs par l'affichage de textures comme suit :

Code

La Fig.16 présente les résultats pour l'application d'une texture brique et l'application d'une texture rocheuse.

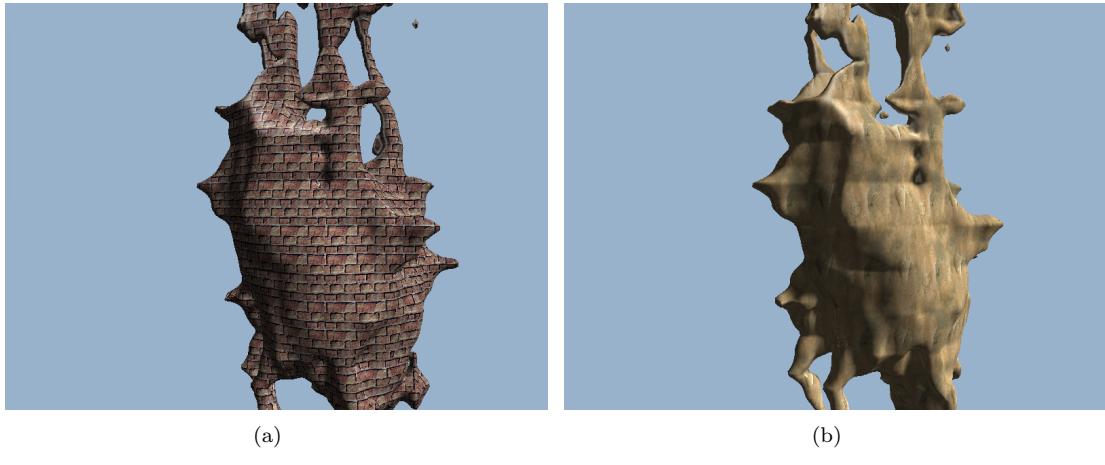


FIGURE 15 – Application des coordonnées de texture par pondération des normales

Choix de la direction de projection relativement aux normales et définition des coordonnées de textures égales aux coordonnées de position :

```

vec2 coord_tex;
vec3 test_color;

if(abs(vf_normal.x) > abs(vf_normal.y) && abs(vf_normal.x) > abs(vf_normal.z))
{
    coord_tex = vf_position.yz;
//    test_color = vec3(1.5*vf_normal.x, vf_normal.y, vf_normal.z); //Ponderation de la
}
else if(abs(vf_normal.y) > abs(vf_normal.x) && abs(vf_normal.y) > abs(vf_normal.z))
{
    coord_tex = vf_position.xz;
//    test_color = vec3(vf_normal.x, 1.5*vf_normal.y, vf_normal.z); //Ponderation de la
}
else
{
    coord_tex = vf_position.xy;
//    test_color = vec3(vf_normal.x, vf_normal.y, 1.5*vf_normal.z); //Ponderation de la
}
test_color = 5*vec3(vf_normal.x, vf_normal.y, vf_normal.z); //Couleur mélangeant les

```

Définition de la couleur d'après les nouvelles normales :

```

float blend_weights = abs(100)-0.2;
blend_weights *=7;
blend_weights = pow(blend_weights, 3);
blend_weights = max(0,blend_weights);
blend_weights /= dot(blend_weights,1);

const float freq = 0.17;
//vec4 texcolor = vec4(1.0,0.,0.,1.0); // Affichage du volume en rouge
//vec4 texcolor = vec4(blend_weights*test_color,1); //Affichage du volume avec couleurs
vec4 texcolor = texture(blend_weights*myTexture, coord_tex); // Affichage du volume avec
...
color = occ_light * occ_light * (blended_color * 0.2 + 0.6 * diffuse + 0.3 * vec4(vec3(

```

Finalement la dernière étape est l'application de l'occlusion de la lumière permettant de savoir quelle quantité de lumière, en chaque point, reste bloqué dans la texture ou parviennent à la caméra. Cela permet un rendu plus naturelle des ombres comme montré Fig.??.



FIGURE 16 – Occlusion de la lumière et rendu final

## 2.6 Combinaison Parallax Mapping + Marching Cubes