

Shaders avancés et Marching Cubes

CPE — ETI5–IMI

octobre 2017

1 Introduction

Lors de ce TP, nous allons utiliser des shaders de manière un peu plus évoluée que pour simplement mimer ce que fait le pipeline fixe ou faire de la GPGPU. Dans un premier temps, nous allons explorer une technique permettant de donner l'illusion du relief là où il n'y en a pas. Puis, nous allons explorer une implémentation de l'algorithme des marching cubes ce qui sera l'occasion de voir un rendu en plusieurs passes (étapes) permettant d'obtenir une surface assez complexe, que l'on texturera de manière complexe.

2 Parallax mapping

Jusqu'à maintenant, nous avons utilisé des textures colorées afin de les appliquer directement sur un maillage. Par exemple, pour dessiner un mur de briques, on pouvait utiliser une texture comme celle de la figure 1

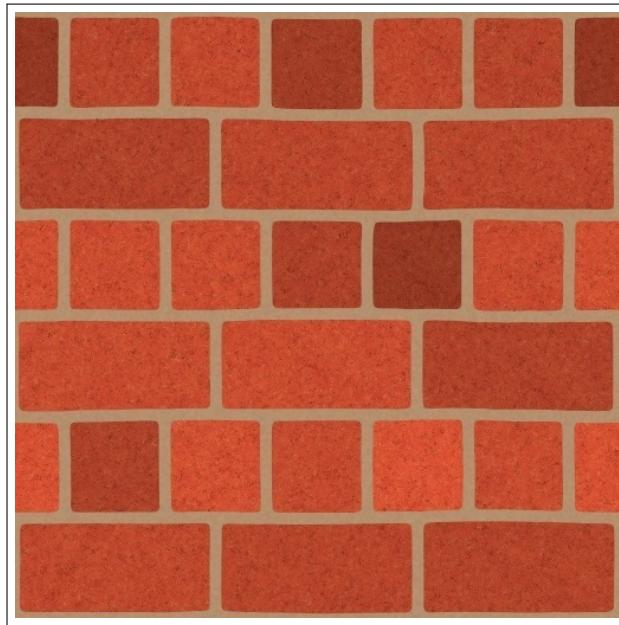


FIGURE 1 – Texture de couleur

Mais les textures peuvent contenir bien d'autres informations que de la couleur.

Dans l'exemple que nous allons étudier, nous utiliserons aussi une texture pour stocker des modifications de normales (figure 2) puis une autre pour stocker des informations de profondeur (figure 3).

Comme vous le savez, on peut utiliser la normale en un point pour calculer une illumination de Phong. Habituellement, cette normale est donnée pour chaque sommet (vertex) puis simplement interpolée pour le calcul de différents produits scalaires. Pour ajouter du réalisme, on peut utiliser la texture de la figure 2 pour modifier cette normale en chaque point (pour une coordonnée de texture donnée). Cette texture contient des différences (géométriques) à appliquer à la normale en chaque point. La composante rouge indique une modification de la normale suivant l'axe x , la composante verte pour l'axe y et la bleue pour l'axe z .

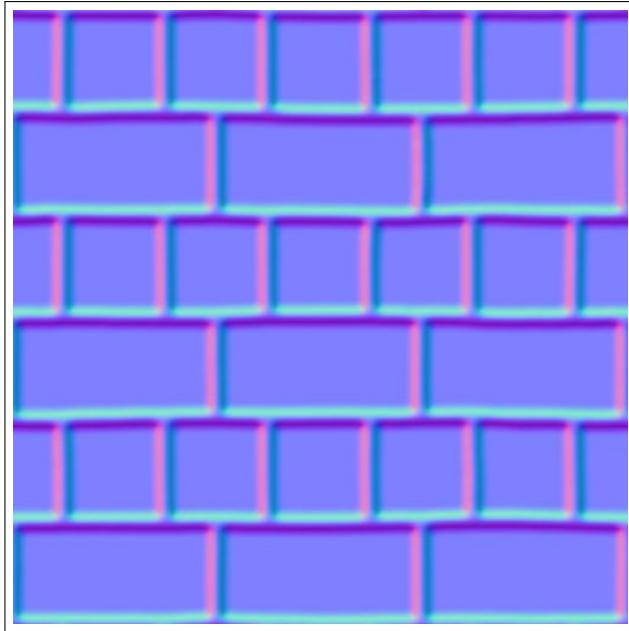


FIGURE 2 – Image utilisée pour modifier les normales

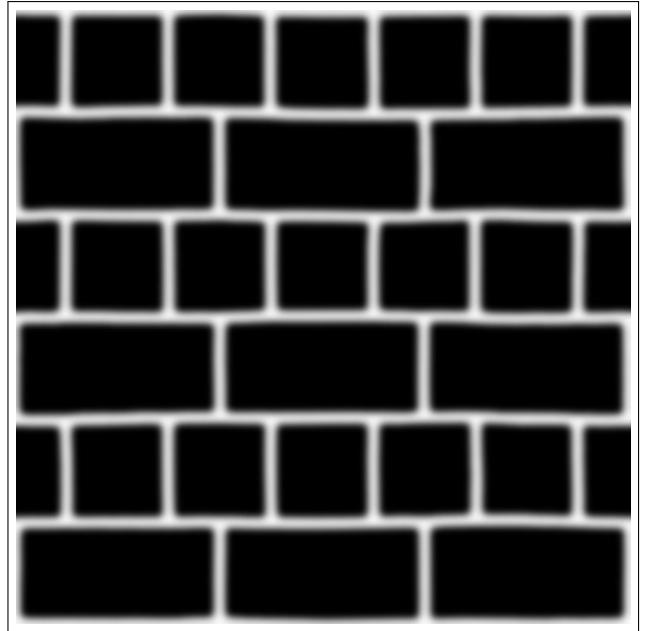


FIGURE 3 – Image des profondeurs pour chaque pixel

Évidemment, cela se calcule dans un repère lié à la surface que l'on dessine. Dans ce repère, la normale « naturelle » est selon l'axe des z (vers nous). Les axes x et y dans ce repère sont tangents à la surface et sont nommés *tangent* et *bitangent*. Il faudra donc passer ces deux nouvelles informations avec chaque sommet (en plus de la position, de la normale et des coordonnées de texture).

Le programme fourni (`parallax.tar.gz`) implémente ce *normal mapping* et permet d'avoir un résultat semblable à ce qui sur la figure 4. On a un éclairage qui utilise une normale par pixel, et donne une impression de relief (notez particulièrement les liserets noirs ou blancs autour de chaque brique).

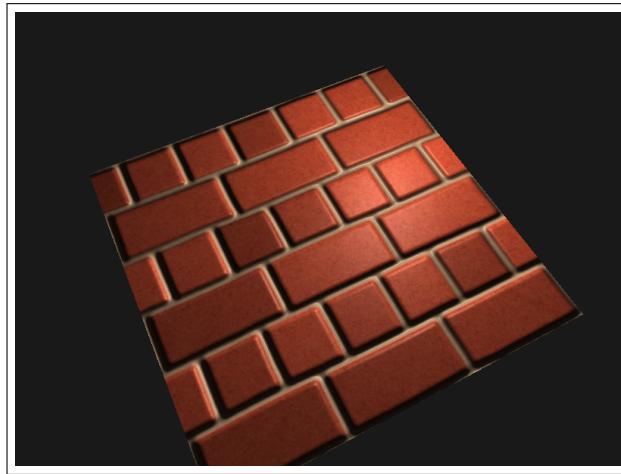


FIGURE 4 – Mur de briques avec Normal Mapping

Question 1 : Compilez et lancez le programme. Remarquez comment la lumière interagit avec la surface.

Question 2 : Analysez le fichier `main.cpp`. Regardez en particulier comment sont calculés les *tangentes* et *bitangentes*. Dans ce cas particulier, ne pourrait-on pas grandement simplifier les calculs ? Regardez aussi comment sont gérées les différentes textures. Quel est le rôle de la fonction `glActiveTexture()` ?

Question 3 : Dans les shaders, assurez-vous de bien comprendre le rôle de chaque variable. Certaines ne sont pas encore utilisées (elles le seront lors de l'implémentation de l'effet parallax), lesquelles ? Dans quels repères sont exprimées les différentes variables `vf_...` ?

Le rendu que permet le *normal mapping* est déjà bien, mais on peut faire mieux. Avec peu de données supplémentaires, il est possible de donner une véritable impression de relief.

L'idée va être d'utiliser une carte d'élévation (*height map*) afin de modifier les coordonnées de textures à utiliser pour les textures de couleurs et de normales.

Imaginons que dans l'espace tangent (*tangente, bitangente, normale*), on regarde depuis la position `view_pos` vers la position `frag_pos` (voir figure 5).

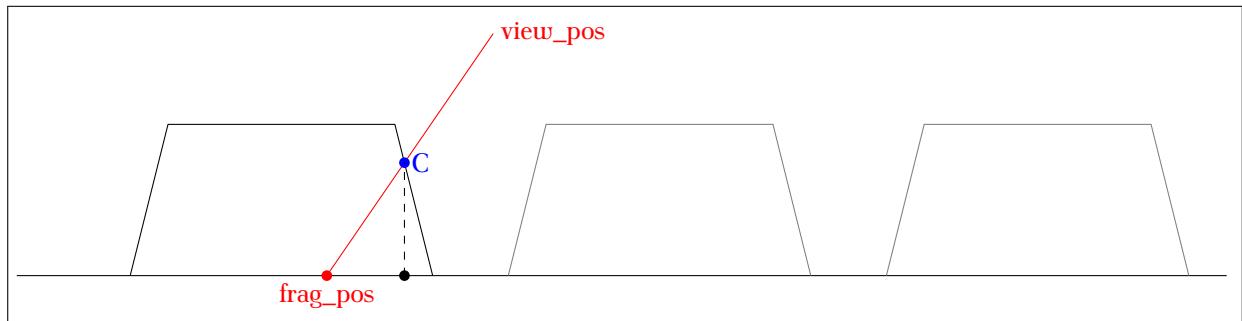


FIGURE 5 – Schéma de fonctionnement du parallax mapping

Idéalement, si l'on voulait avoir un véritable effet de relief, il nous faudrait trouver le point C de la figure, et faire nos calculs à partir de là. Malheureusement, ce point est assez complexe à obtenir exactement. On va donc l'approximer en disant qu'il est sur la demi-droite $[frag_pos, view_pos]$, et à une distance proportionnelle à la hauteur de la texture d'élévation à la position correspondant à `frag_pos`. Cette approximation est assez grossière, surtout avec des hautes fréquences dans la texture d'élévation, mais elle donne des résultats assez convainquants. De plus en supposant le vecteur $\overrightarrow{view_dir} = \overrightarrow{frag_pos} - \overrightarrow{view_pos}$ normé, diviser par la composante z améliore encore les résultats en donnant un relief d'autant plus important que la vue est rasante. (pourquoi ?)

On a alors l'algorithme suivant :

- Récupération de la hauteur en utilisant la texture d'élévation et les coordonnées de base
- Calculer le vecteur `view_dir`
- Décaler les coordonnées de texture proportionnellement à la hauteur trouvée et à la projection de `view_dir` sur le plan de la surface
- Le décalage peut prendre en compte la division par `view_dir.z` ou pas
- Utiliser les nouvelles coordonnées de textures avec les textures de couleurs et de normales pour calculer la couleur finale comme dans le cas du *normal mapping*.

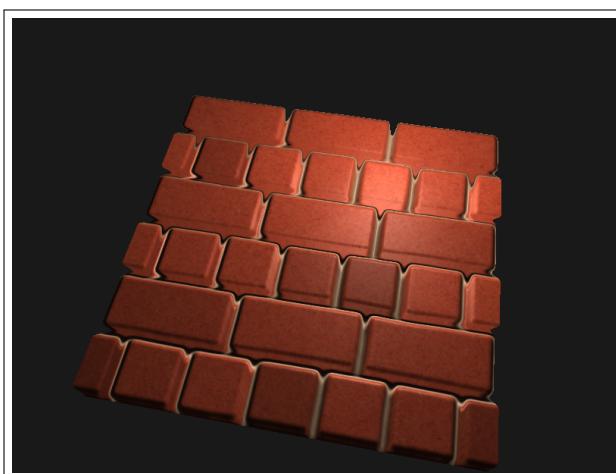


FIGURE 6 – Mur de brique avec Parallax Mapping

Question 4 : Modifiez `main.cpp` et `parallax.frag` afin d'implémenter l'algorithme de parallax et d'obtenir une image semblable à la figure 6. N'hésitez pas à ajouter des moyens de paramétriser le rendu avec les touches du clavier

Note :

Cette section est largement inspirée de <http://www.learnopengl.com/#!Advanced-Lighting/Parallax-Mapping>

Pour aller plus loin avec ce genre de techniques : http://http.developer.nvidia.com/GPUGems3/gpugems3_ch18.html

3 Marching Cubes

Dans cette partie, on se propose d'étudier et d'améliorer une implémentation de l'algorithme des *Marching Cubes* présenté en cours.

L'archive compressée fournie (*marching-cubes.tar.gz*) permet le calcul et l'affichage de triangles formant la sphère visible sur la figure 7.

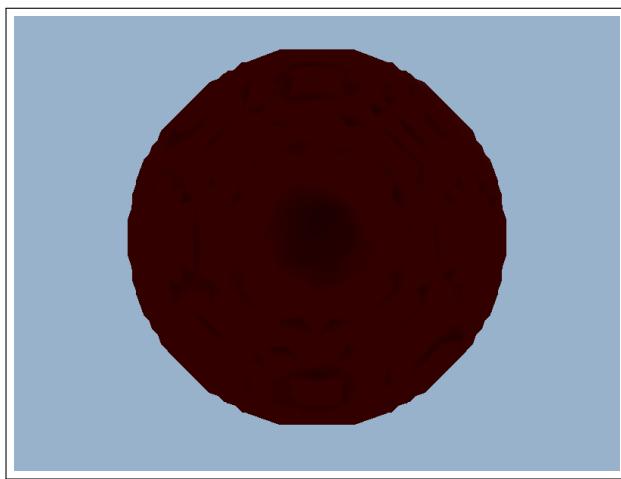


FIGURE 7 – Une sphère assez moche

3.1 Présentation générale

Quatre grandes étapes qui utilisent des techniques assez avancées comme les *framebuffers*, les *transform feedbacks* ou les *geometry shaders* :

- **Création d'une fonction de densité.**

Cette étape utilise un *fragment shader* pour écrire dans un *framebuffer* lié à une texture 3D. Lors de cette étape, le *fragment shader* est appelé pour chaque position 3D et doit fournir une valeur de densité dépendant de la position.

- **Liste du nombre de triangles et des intersections correspondantes.**

Cette étape implémente une partie de l'algorithme des *Marching Cubes* en utilisant les tables présentes dans le fichier *marching_cubes.cpp*. Son rôle est juste de stocker les numéros des arêtes contenant les intersections pour chaque cube.

- **Génération des triangles.**

C'est dans cette étape que sont calculées les positions, normales et occlusion ambiante (pour l'éclairage) des sommets. L'algorithme est découpé en deux étapes pour éviter les calculs inutiles pour les sommets non utilisés. Vous trouverez plus de détails ici : https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch01.html dont ce TP est largement inspiré.

- **Rendu des triangles.**

Cette étape, qui est la seule à devoir être répétée à chaque affichage effectue simplement un rendu des triangles obtenus à l'étape précédente.

Par ailleurs, la zone du monde parcourue par l'algorithme est découpée en $3 \times 3 \times 8$ blocs, chacun contenant $48 \times 48 \times 48$ cubes élémentaires.

Question 5 : Repérez quel groupe de shaders et quelle partie du code C++ sont liés à chaque étape. Essayez de faire un schéma indiquant comment les différentes étapes échangent des données via des FBO ou des TF.

3.2 Analyse du code

Le code de cette partie utilise les bibliothèques mathématiques habituelles (dans `lib/`), des shaders et deux fichiers C++ dans `src/`.

Certaines fonctions utilisées (OpenGL ou GLSL) n'ont pas été présentée en cours, mais vous pouvez évidemment les trouver dans la documentation officielle, sur <http://opengl.org/>.

Prenez le temps de bien comprendre l'intérêt de ces fonctions.

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 ?

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

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` ?

N'hésitez pas à poser des questions si certaines parties du code sont obscures !

3.3 Amélioration de l'algorithme

Le code fourni n'est pas optimal. Notamment, les normales ne sont pas calculées correctement (la même normale est renvoyée pour tous les sommets).

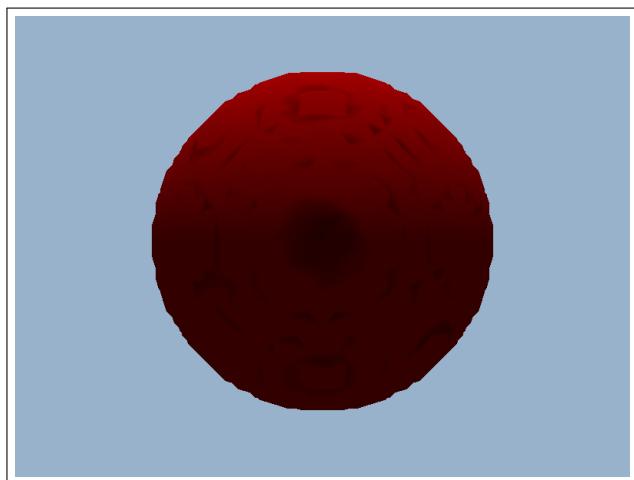


FIGURE 8 – Une sphère toujours moche (mais moins)

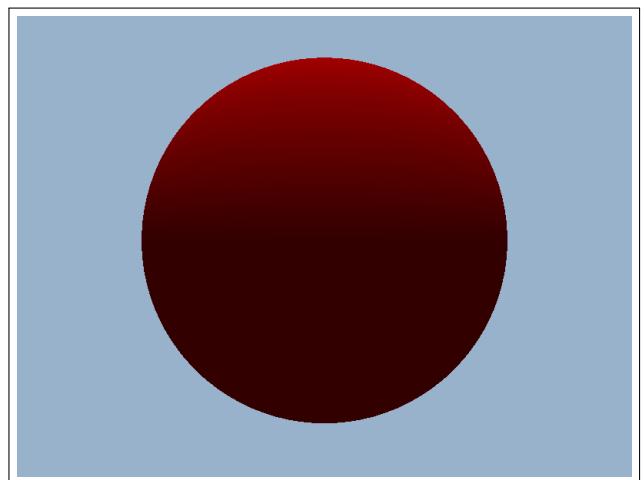


FIGURE 9 – Une belle sphère bien lisse

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 ?

L'aspect crénélisé de la sphère vient du fait que la position des vertex est toujours choisie au milieu des arêtes des cubes. Une meilleure approche consiste à réaliser une interpolation linéaire pour choisir cette position.

Question 10 : Réalisez cette interpolation de manière à obtenir une sphère bien lisse comme celle de la figure 9.

3.4 Fonctions de densité

La fonction de densité proposée permet d'obtenir une sphère, ce qui est l'une des formes les plus simples. Cependant, cette forme n'est pas vraiment passionnante. En changeant la fonction `density`, il est possible de définir toutes sortes de formes.

Par exemple, on obtient facilement un cylindre (figure 10) ou une boîte (figure 11).

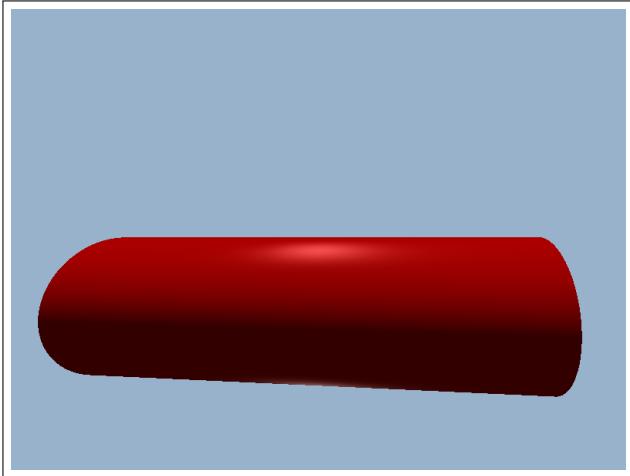


FIGURE 10 – Un cylindre

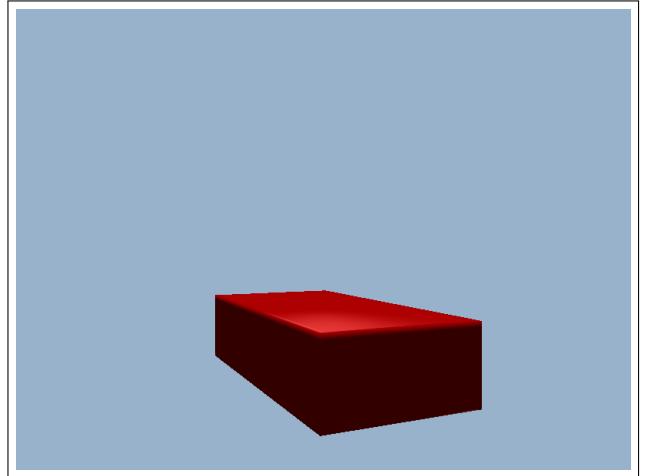


FIGURE 11 – Une boîte (un cube)

Vous trouverez des exemples de fonctions intéressantes sur cette page :
<http://iquilezles.org/www/articles/distfunctions/distfunctions.htm>

Mais pour la suite du TP, nous allons avoir besoin de forme plus complexes comme celle des figures 12 à 15.

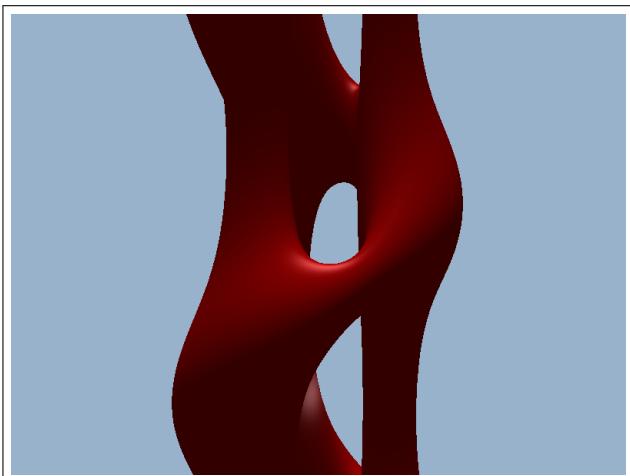


FIGURE 12 – Une structure plus intéressante

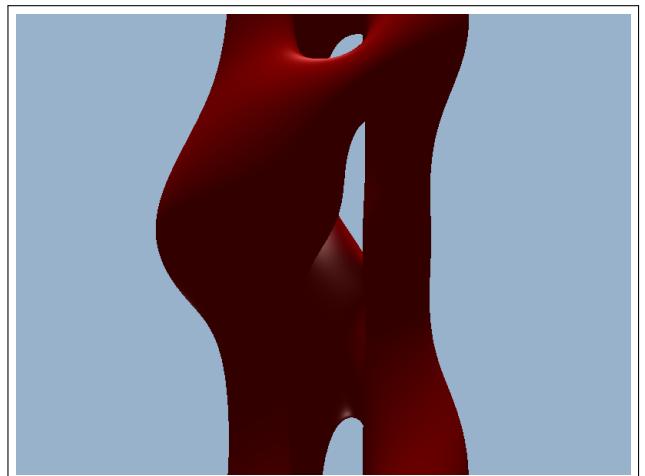


FIGURE 13 – Une autre

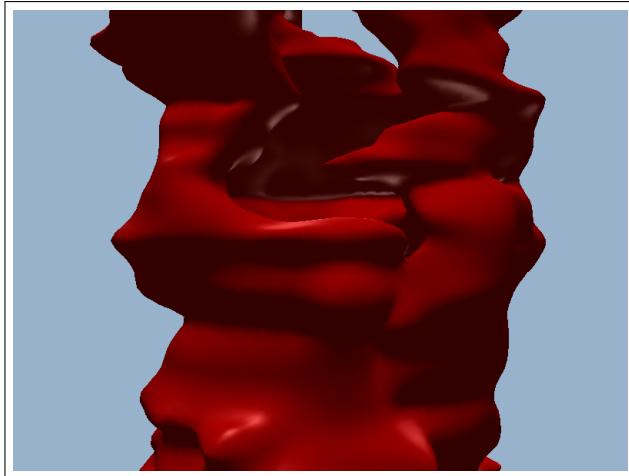


FIGURE 14 – Avec un peu de bruit

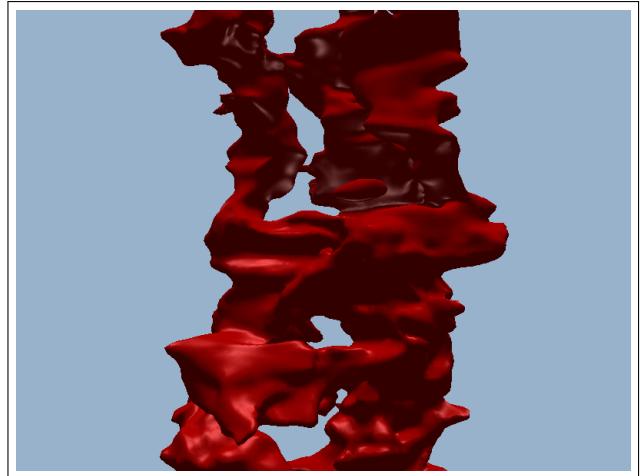


FIGURE 15 – Avec plus de bruit

Ces formes sont inspirées du début du document : <http://dindinx.net/cpe/eti5/opengl-gpu-webgl/CascadesDemoSecrets.pdf>. Vous êtes bien entendus libres de choisir bien différentes.

Question 11 : Créez une fonction de densité aussi intéressante que possible

3.5 Améliorations graphiques & Textures

Maintenant que vous avez une forme qui vous intéresse, il est temps de lui donner le meilleur aspect possible.

L'une des façons les plus évidentes consiste à appliquer une texture. Cependant, pour cela, il est nécessaire de disposer de coordonnées de texture, hors nous n'avons que les positions et les normales des sommets, et il serait impossible d'avoir des coordonnées de texture cohérentes étant donné que notre forme est quelconque. Il nous va donc falloir "inventer" de telles coordonnées de texture.

L'une des manières est d'utiliser les coordonnées spatiales (les positions) comme coordonnées de texture, en n'utilisant que deux dimensions (par exemple x et y). Cela semble faire l'affaire, mais les résultats ne sont pas terrible (voir figure 16).

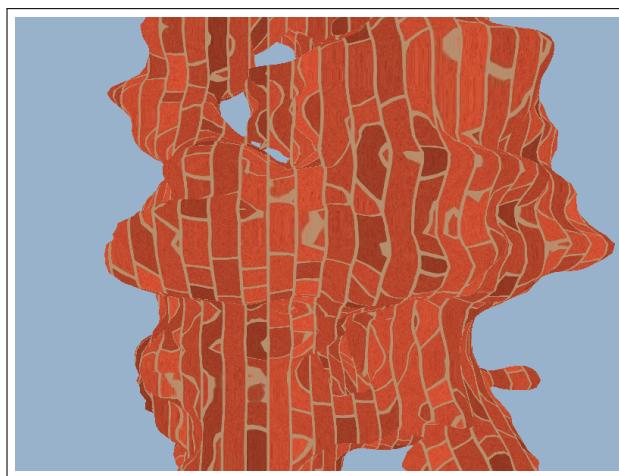


FIGURE 16 – Bien mauvaise application de coordonnées de texture

Le problème vient du fait que l'on privilégie une seule direction : si on utilise x et y comme coordonnées de texture, la texture est projetée suivant z . On peut alors choisir x et z ou y et z pour avoir d'autres directions privilégiées, chacune ayant plus ou moins d'intérêt suivant l'orientation de la forme.

Une manière d'améliorer les choses consiste à mélanger ces différentes vues, en utilisant la normale en chaque point. Ainsi,

si la normale est plutôt suivant l'axe des y , on utilisera les coordonnées spatiales (de position) x et z comme coordonnées de textures, et respectivement sur les autres axes. Pour avoir des transitions douces, on utilisera simplement chaque composante de la normale comme "poids" venant pondérer les couleurs obtenues en échantillonnant la texture suivant les différents axes.

Il est possible de modifier chaque normale afin de donner un poids plus important à la direction principale, cela rend le rendu plus naturel.

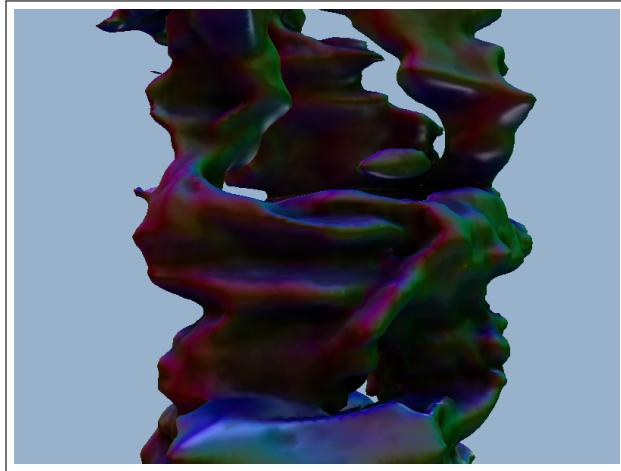


FIGURE 17 – Visualisation des normales

Vous pouvez utiliser ces normales modifiées comme couleur, pour vérifier que vos calculs sont bons, comme sur la figure 17.

Question 12 : Mettez en place le calcul pondéré des textures par les normales pour obtenir une image du genre de la figure 18.



FIGURE 18 – Meilleure application de texture



FIGURE 19 – Autre rendu

Vous pouvez évidemment améliorer la qualité du rendu de votre forme autant que vous le souhaitez.

On peut par exemple imaginer utiliser le *parallax mapping* de la première partie de ce TP pour ajouter de la profondeur à la forme. Soyez créatif!