Manal BENAISSA - Nadia BENMOUSSA

TP09 :Projet de synthèse d'image

Rendu d'un terrain en utilisant le bruit de Perlin
Avril 2019

Introduction :

Ce projet consiste à rendre un terrain en utilisant le bruit de Perlin et les différentes technique vues dans le cours.
Voici les étapes que nous avons réalisé:

La première passe : Texture de Perlin

A savoir :
Chaque texture possède son propre FBO (par pur commodité). Ainsi, un tableau d'FBO a été créé, contenant :

- fbo[0] : la texture de Perlin généré par les shaders noise.vert/frag (fournis)
- fbo[1] : la texture des normales
- fbo[2] : la texture de profondeur

Il existe cinq pairs de shaders :

- noise.vert/frag
- normal.vert/frag
- verifFBO.vert/frag
- terrain.vert/frag
- first-pass.vert/frag

La première passe consiste à créer la texture de Perlin et à la stocker dans une FBO (ici dans fbo[0]). De manière général, la création/affichage de la texture suit ce modèle :

```
glViewport(0,0,512,512);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
fonctionQuiActiveLeShaderAssocie();
drawVAOCarre(); //dessin de la géométrie (carré) qui servira de support à la texture
disableShader(); //Va désactiver TOUS les shaders
```

Cette partie se trouve dans **PaintGL()**, qui gère ainsi l'activation des shaders, des VAOs et des FBO. A celà, on ajoute (pour chaque étape) une

fonction d'activation de shaders, et qui se chargera éventuellement de transmettre les variables (uniform) ou les textures nécessaires.
La deuxième étape consiste à placer cette nouvelle texture (qui apparaît à l'écran) dans une FBO, et à TESTER qu'il est bien dans le FBO, en réaffichant son contenu. Ainsi, le code ci-dessus est complété comme suit :

```
 glDrawBuffer(GL_COLOR_ATTACHMENT0);
 glBindFramebuffer(GL_FRAMEBUFFER,_fbo[numDeFBOaTextureAssocie]); //on active le mode "écriture en buffer"
 glViewport(0,0,512,512);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 fonctionQuiActiveLeShaderAssocie();
 drawVAOCarre();
 disableShader();

 glBindFramebuffer(GL_FRAMEBUFFER,0); //On désactive le mode "écriture en buffer"

 //Vérification qu'on a bien notre FBO (Normal) (à remettre en commentaire quand c'est vérifié !)
 glViewport(0,0,512,512);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 enableShaderVerifFBO(_textureATester); //On va utiliser le shader qui affiche le FBO (verifFBO.vert/frag)
 drawVAOCarre();
 disableShader();
```

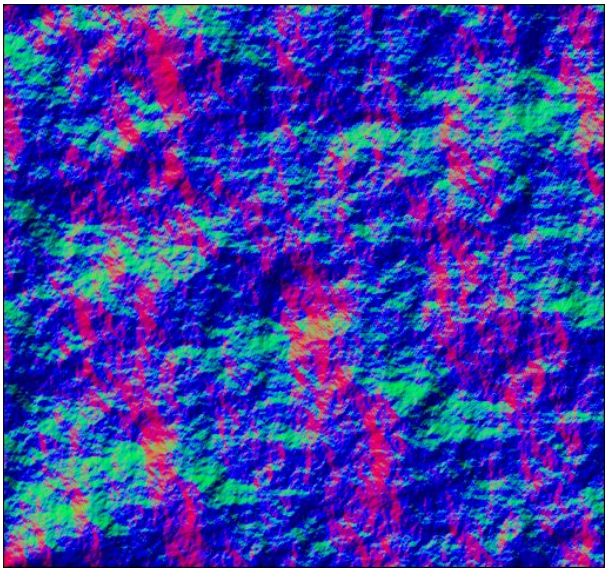
A la fin de la 1ère passe, la texture de Perlin est générée et stockée en FBO. Ce qui nous permet de passer à la seconde passe…

La deuxième passe : Texture des Normales

Cette passe ressemblera sensiblement à la 1ère : une fonction va activer les shaders normal.vert/frag, et lui transmettra la texture de Perlin (uniform). Le shader normal.vert ne fait rien, et normal.frag récupère la texture et calcule la normale à chaque pixel. Ces normales

de coordonnées (x,y,z) sont restituées dans la texture sous forme de couleurs (r,g,b). Cette texture est ensuite stockée dans fbo[1].

Nota Bene : Il aurait été sans doute préférable de stocker toutes ces textures dans un seul FBO, puisqu'elles ont la même résolution. Il aurait suffit de diriger la texture de Perlin dans **GL_COLOR_ATTACHEMENT0** et celle des normales dans **GL_COLOR_ATTACHEMENT1.** Cela aurait certainement facilité la suite…



Texture de normales obtenue

Ceci conclut la 2ème passe.

La troisième passe : Création du terrain

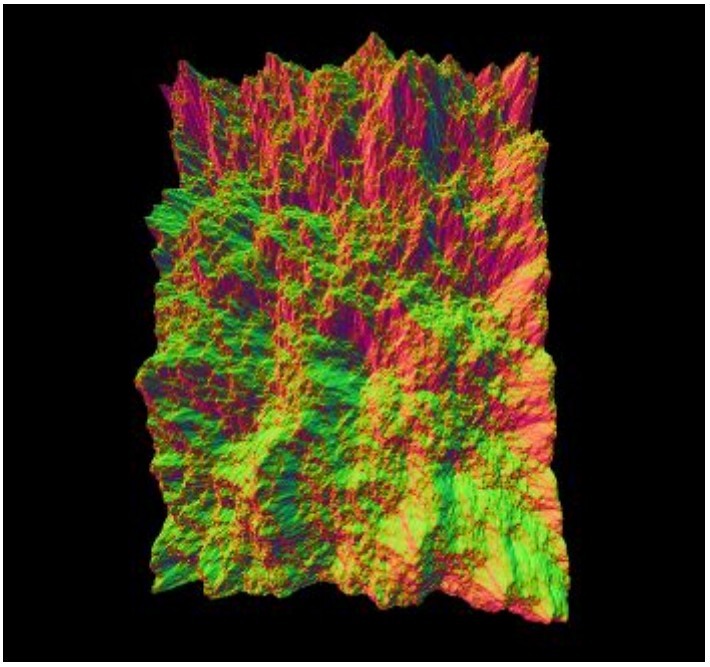
Une fois les textures de normales et de perlin créés, il faut à présent créer le terrain. Une fonction toute faite permettait de dessiner le VAO associé. Il faut une fonction pour activer les shaders terrain.vert/frag, à qui on enverra les données suivantes :

- La matrice Modèle-Vue
- La matrice de Projection
- La matrice des normales de la CAMERA
- La texture de Perlin
- La texture des normales (du TERRAIN)
- Le vecteur de direction de la lumière
- (facultatif) un entier pour l'animation, qui représentera le temps

Le vertex shader se charge ensuite de récupérer la texture de perlin, et d'associer la valeur de chaque point de la texture avec la coordonnée z des sommets du terrain. Ainsi, nous passons d'un terrain

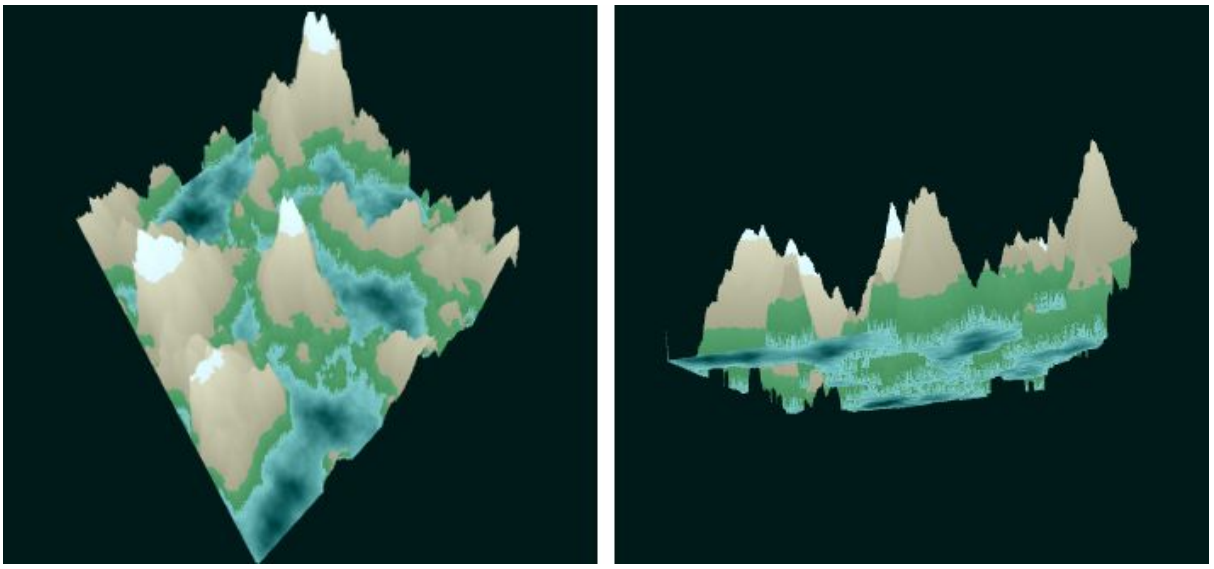
plat à un terrain montagneux. Il est également possible à ce moment là de réajuster la structure même du terrain (adoucir les montagnes, éviter les reliefs en dessous du niveau de la mer etc…). **De manière général, tout ce qui touchera la structure du terrain se fera dans terrain.vert.**

Ensuite, nous pouvons passer aux couleurs du terrain dans le fragment shader terrain.frag : C'est ici notamment qu'on ajoute le blanc pour la neige, le brun pour la terre, le vert pour l'herbe et le bleu pour l'eau. Nous aurions pu chercher une texture sur internet pour imiter le terrain, nous avons d'ailleurs testé de coller une texture lambda sur nos montagnes :



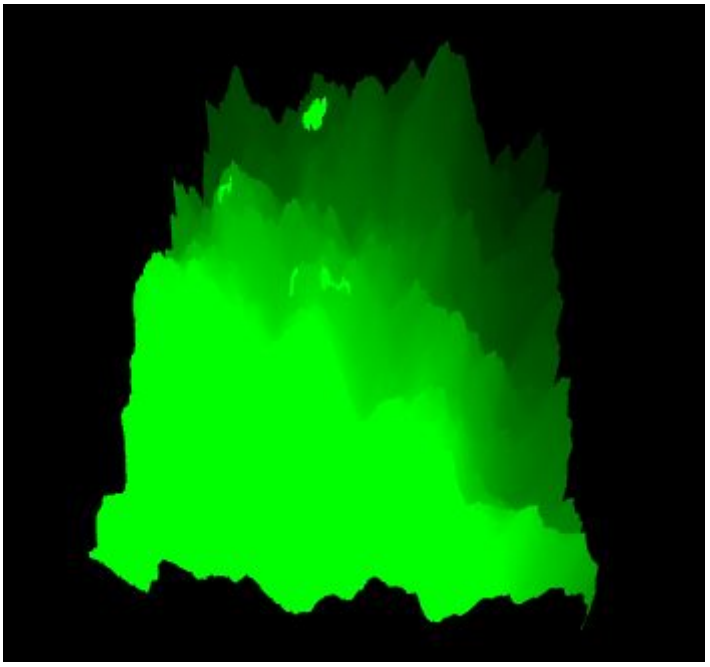
Ajout d'une texture (ici la texture de normales, mais on aurait pu faire ça avec n'importe quelle texture ~)

Finalement, c'est ce format qui a été retenu :



Nous avions trouvé l'effet sur l'eau particulièrement esthétique.

L'étape suivant consiste à ajouter de la lumière. Les données nécessaires pour ça ont déjà été transmises au fragment shader. Des tests ont été fait auparavant, nous menant à ce résultat (peu esthétique) :

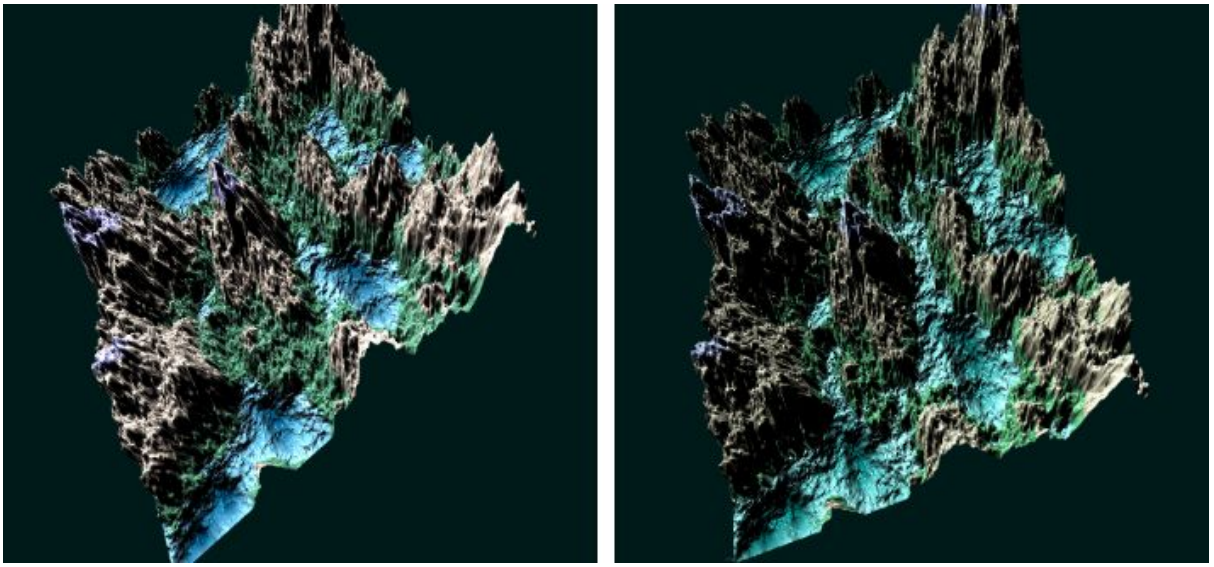


Les normales n'étaient pas encore transmises à ce moment là, on testait que notre lumière fonctionnait bien.

```
//PARAMETRE DE PHONG
      float et = 10.0;
      vec3 n = normalize(normalView);
      vec3 e = normalize(eyeView);
      vec3 l = normalize(light);
```

```
//PHONG MODEL
      float diff = max(dot(l,n),0.);
      float spec = pow(max(dot(reflect(l,n),e),0),et);
      vec4 colorMountain =
choisirCouleur(texture(textureAAfficher,uvcoord)); //couleur des
montagnes (neige,eau...)
      outBuffer =colorMountain*(diff + spec)*1.5;
```

La gestion de la lumière suit le modèle de Phong, comme présenté ci-dessus. Il est effectué dans le fragment Shader et permet d'obtenir le résultat suivant :



Lorsque la lumière se déplace, les ombres suivent sur les montagnes. Le rendu général est "agressif", car les montagnes sont très escarpées. L'idéal aurait été d'adoucir l'ensemble.

Nous pouvions passer à la quatrième passe suite à ça.

La quatrième passe : Des effets !

Nous voulions ajouter du brouillard, et pour cela, nous devions récupérer la profondeur. Pour cela, nous avions voulu suivre à peu près le même schéma que les autres passes, à savoir :

- Créer un shader pour récupérer la profondeur : first-pass.vert/frag
- Transmettre cette valeur au terrain.frag qui calculera ensuite le brouillard.

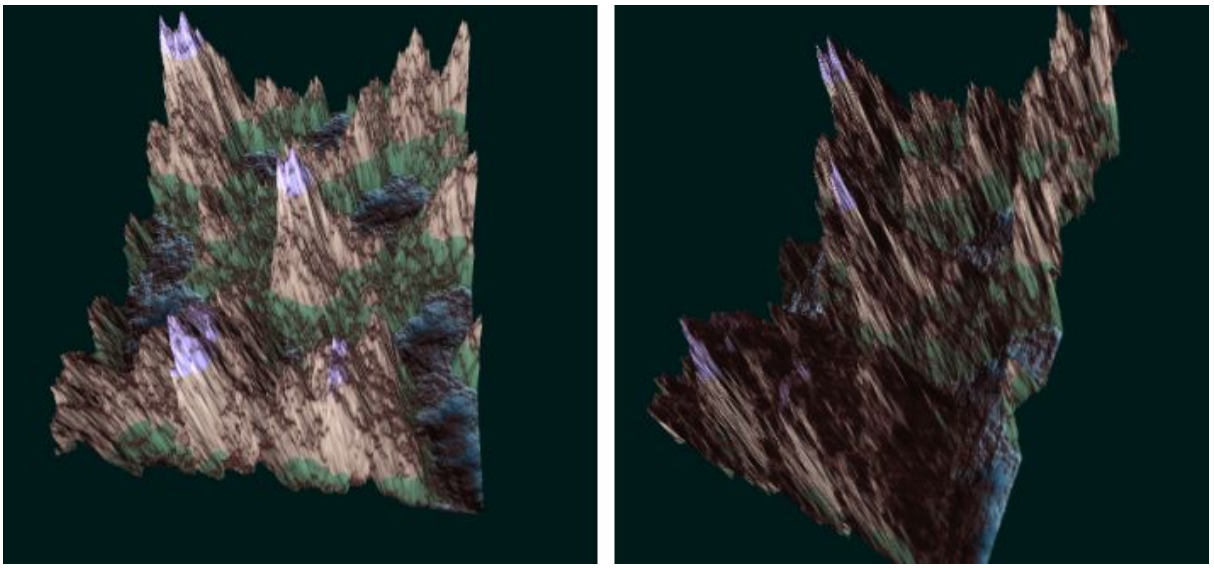
Le calcul du brouillard dans le terrain.frag s'est fait de la façon suivante :

```
  vec4 shade(in vec2 coord) {
  vec4  nd = texture(texNormal,coord,0);
  vec4 colorMountain =
choisirCouleur(texture(textureAAfficher,uvcoord)); //couleur des
montagnes (neige,eau...)
  vec4 c =colorMountain;

  vec3 n = normalize(normalView);
  float d = nd.z; //Profondeur qu'il faut récupérer avant dans
first-pass.vert/frag

  vec3 e = vec3(0,0,-0.2);
  vec3 l = normalize(light);

  float diff = max(dot(l,n),0.0);
  float spec = pow(max(dot(reflect(l,n),e),0.0),d*10.0);

  //AJOUTE : S est pour "shade" et prend en compte l'ombrage
  vec4 S = colorMountain*(diff + spec)*2.0;
  // AJOUTE : F pour "Fog", et donne la couleur du brouillard
  vec4 F = vec4(vec3(0.2,0.1,0.1),1);

  //AJOUTE : On mix les deux filtres, "clamp" permet de mettre d
(la profondeur) entre 0 et 1
  return mix(S,F,clamp(0,1.,d));
}
```

L'étape suivante consistait donc à récupérer la profondeur **d** en faisant une première passe dans le shader **first-pass.vert/frag**, puis à la transmettre à la fonction ci-dessus pour avoir un joli brouillard.

Malheureusement, nous n'avons pas su faire cette étape : En effet, nous étions parties pour suivre le même schéma que les autres passes : créer une texture de Profondeur, la mettre dans **fbo[2]** et la réutiliser dans **terrain.vert/frag**. Cette texture était néanmoins "spéciale" (et ne semblait pas pouvoir être convertie en texture classique) : Lorsque nous affichions la texture directement à l'écran, elle changeait selon la position de la caméra. Une fois stockée dans le FBO et revisualisée ensuite (avec **verifFBO.vert/frag**), nous obtenions une texture bleue basique.

Cela donnait le rendu suivant sur le terrain :



Peu visible sur ces images, nous avions bien un "brouillard", mais qui ne dépendait pas de la position de la caméra.

Nous n'avons pas eu le temps de régler ce problème hélas.

Conclusion :

De manière général, ces étapes ont permis d'avoir un terrain en relief, avec de la lumière et un pseudo-brouillard. Nous regrettons de ne pas avoir eu plus de temps pour finir proprement le brouillard et ajouter l'ombre, et éventuellement d'autres effets sympathiques (nous étions très motivées pour ça). Globalement, le projet a été très ludique et instructif, bien qu'assez corsé à certains moments. Mais l'utilisation des FBO, des shaders, et la structure globales du viewer.cpp ont été bien compris (vous trouverez d'ailleurs une floppée de commentaires). Le travail s'est fait en binôme de façon équitable et harmonieuse, sur une même machine. Vous trouverez même un git, en cas où : https://github.com/hirouka/SIProjet