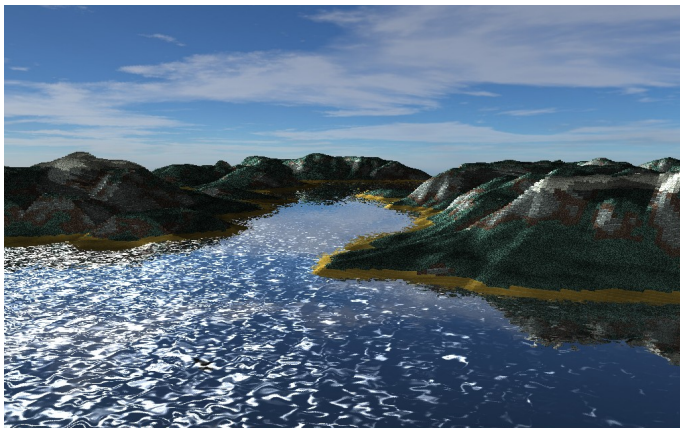
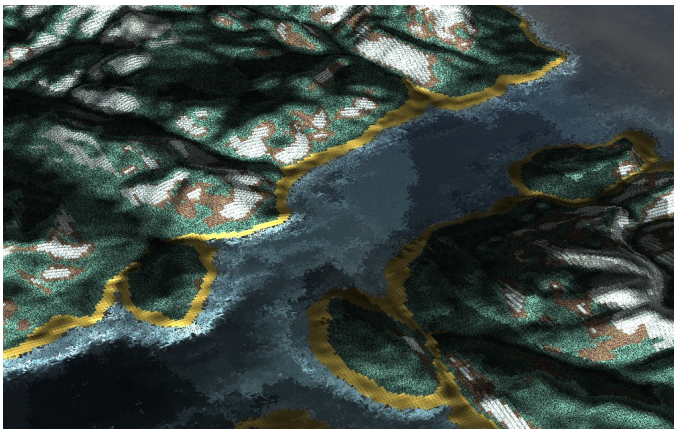
Compte-Rendu de projet

UE : Synthèse d'Image 3D

Monôme : Peurière Romain

Le projet consistait à nous faire manipuler le pipeline d'openGL dans le cadre d'une voxellisation d'un terrain représenté par une image en grayscale (HeightMap). Imitant ainsi un rendu à la « Minecraft ».





Liste des fonctionnalités

Comme demandé dans le(s) sujet(s), les fonctionnalités suivantes sont implémentées dans le projet :

- Chargement du terrain (TP1)
- Séparation du terrain en régions (TP2 – Partie 1)
- Test de visibilité bidirectif des régions (TP2 – Partie 2)
- Séparation des régions en matières (tris sur les données du terrain) (TP2 – Partie 3)
- Plusieurs passes de rendu
  - ShadowMap (TP2 – Partie 4)
  - Réflexion de l'eau
  - Réfraction de l'eau
  - Draw général
- Ajout d'une cubemap

Commandes :

Manipulation de la caméra : ZQSD (centre) et Clics gauche (angles)
Affichage des régions : touche « l »
Affichage des textures (debug) : touche « k »

Paramètres :

Taille des régions, Nombre de régions, Nombre de matières (code)

Choix d’implémentations

Implémentation globale :

Comme indiqué dans le TP2, nous devions faire un choix d’implémentation au niveau du stockage des données et donc implicitement de l’affichage :

- Avoir 1 buffer & VAO général (Scène) (Par matière, pour chaque région)
- Avoir 1 buffer & VAO par région (Région-indépendant) (Par région, pour chaque matière)

Le but étant de minimiser le nombre d’appels aux fonctions openGL, il m’a paru optimal d’opter pour la première solution.

Effectivement, bien que nous séparions nos régions pour la visibilité, nous n’affichons que (ou presque) des cubes, ainsi toute région est ‘similaire’ et facilement descriptible par un draw instancié.

Il paraît alors plus judicieux d’opter pour un seul buffer & vao général. Si nous rajoutons l’élément des textures et donc du bind des textures ainsi que des shaders (opération longue), il paraît d’autant plus évident de minimiser ces opérations et d’ainsi ne les produire qu’une fois par unité, et non une fois par unité par région. Nous n’aurons ainsi qu’à manipuler des indices correspondant aux quantités de données dans le buffer global, plutôt que de manipuler des buffers & shaders eux-mêmes.

Classes :

Pour ce faire, plusieurs classes ont été développés afin de faciliter cette manipulation de donnée.

Terrain/Region Paramètres : Nombre de régions/Taille des régions

La classe terrain a pour but de stocker toutes les informations nécessaires à l’affichage de celui-ci, ainsi que d’effectuer tous les traitements allant de sa création (lecture de la heightmap) jusqu’au remplissage des buffers de positions etc.

Celle-ci a d’abord été codée comme une classe générique (non-GL) pouvant extraire les hauteurs de l’image et fournir toute donnée en tout point (gradient, normale au point, hauteur, position….).
Par la suite, celle-ci a été enrichie pour son implémentation openGL (remplissage des buffers etc). Il aurait tout à fait été possible de dériver cette classe à cet effet.

Ainsi un terrain se compose principalement d’une liste de régions et d’une liste de hauteurs extraites de l’image. Chaque région renseigne les données nécessaires à l’affichage : bounding box, taille du buffer occupée, indice de commencement dans le buffer, et les séparations pour chaque matière/texture (nbmat+2 int). Ce terrain permet la gestion des régions (affichage et tests de visibilité).

Code :

Afin d’implémenter la séparation des matières, après lecture des hauteurs, la fonction fillBuffers() rempli les tableaux de positions normales et pentes en effectuant 2 tris (std::sort), l’un sur la hauteur, sur chaque région, puis sur la pente (entre 2 matières ex : entre l’herbe et la terre, si il y a trop de pente, on met de la terre, sinon de l’herbe).

Afin de limiter le temps d’exécution des swap du tri, le tri ne se fait uniquement que sur les indices du tableau de hauteurs, et est par la suite analysé pour créé les données des buffers (sans oublier les cubes en hauteur : 2 indices identiques consécutifs signifie que le cube actuel se situe sous le précédent).

Framebuffer/Textures/Camera

Ces classes aux noms explicites ont été créées afin de faciliter la manipulation des fonctions openGL ainsi que de la caméra pour l'affichage. Celles-ci permettent de créer un FBO / Une texture, le bind, debind etc. J'ai hésité entre implémenter uniquement des fonctions génériques pour bind etc ou de vraies classes. N'ayant pas croisé de classes les représentant sur des tutoriels etc, j'ai tenté l'expérience : cela paraît optimal pour le projet, mais difficilement réutilisable pour d'autres si l'on veut effectuer d'autres opérations compliquées. Bien que celles-ci n'empêchent à priori pas l'accès aux données d'une quelconque façon, je suis conscient qu'il n'est probablement pas d'usage de créer des classes pour les objets openGL.

Utility

Classe contenant diverses fonctions génériques comme le test de visibilité (effectué dans terrain.cpp) entre 2 bounding boxs (dans le repère du monde, et le repère projectif).

Main

Classe de l'application openGL créant le contexte, avec le code de l'init et le render.

Affichage :

Comme précisé précédemment, 4 passes de rendu ont lieu pour notre terrain.

1. Shadow Map

Comme requis, une passe de rendu a été ajoutée afin de produire une shadowMap et ainsi détecter quels cubes se situent à l'ombre pour le modèle de luminance. Nous pouvons noter que celle-ci sera utilisée par toutes les autres passes et est ainsi la première effectuée.

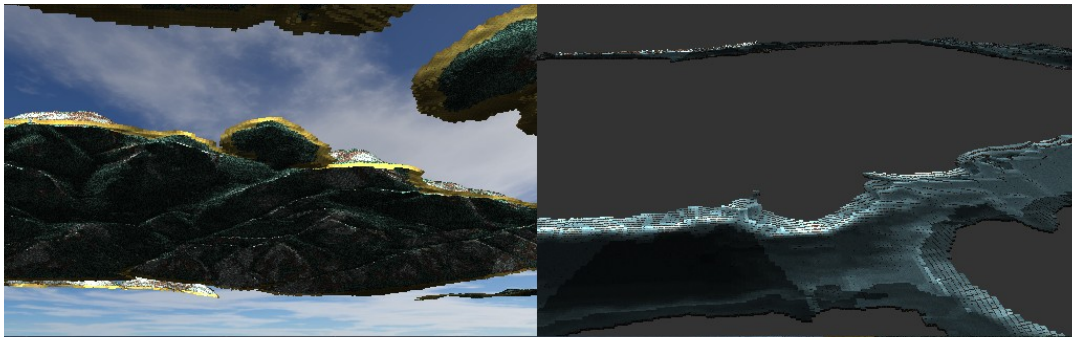
Bien qu'utilisant le même shader pour toutes les matières, n'ayant ainsi pas créé de modèle de réflectance différents, je me suis attelé à créer un effet plus sympathique concernant l'eau. Celle-ci applique donc l'effet de Fresnel grâce à 3 passes de rendu :

2. Effet de l'eau - Réflexion

La première passe consiste à produire une texture du terrain pour l'image de réflexion. Pour celle-ci nous devons faire le rendu de tout ce qui se situe au-dessus de l'eau, et devons ainsi modifier la caméra pour la situer symétriquement inverse par rapport à l'eau (sous l'eau, d'où la création d'une classe camera personnelle). Nous pouvons noter que notre structure d'affichage par matière permet de se passer de l'utilisation de plans de clipboard pour n'afficher qu'une partie du terrain (l'utilisation des clipboard ralentit beaucoup l'affichage).

3. Effet de l'eau - Réfraction

La deuxième passe fait l'inverse : pour la texture de réfraction nous avons besoin de tout ce qui se situe sous l'eau.



Reflection texture refraction texture

4. Draw global

La dernière passe consiste à projeter ces textures de manière correcte sur un plan représentant l'eau (il ne sera ainsi plus nécessaire de draw ce qui se situe dessous, étant opaque). Le tout mélangé proportionnellement à l'angle de vision (Fresnel). Une distorsion grâce à une texture du-dv et un facteur de mouvement ainsi qu'une perturbation des normales avec l'image associée sont appliqués pour un meilleur rendu.

De la même façon, lors de la dernière passe de rendu avec l'eau, l'ensemble du terrain se situant au dessus de l'eau est dessiné (chose facile grâce à l'affichage par matière, et donc par hauteur suite au tri initial).

NB : L'affichage des textures de réfraction/réflexion pour debug est toujours disponible avec la touche 'k'.

Shaders :

Différents shaders ont été écrits pour les besoins de ce projet.

-Le shader principal affichant les cubes, effectue un blinn-phong et altère la lumière suivant la pente, les ombres et la profondeur pour les cubes sous-marins. Il aurait été possible de séparer un shader par matière, mais très peu utile ici ayant le même code, si ce n'est éventuellement la proportion de lumière diffuse dans la formule (un uniform ?) .

-Le shader pour l'eau effectuant la projection des textures, le distortion tiling avec la dudvmap et ajout de la normalmap le tout avec blinn-phong.

-Un shader d'affichage 2D pour le debug des textures

-Un shader basique pour la shadowmap

-Les shaders récupérés des tutos : text et cubemap.glsl

NB : Dans le fichier mainShader.glsl, j'ai essayé d'implémenter le modèle de B.Walter pour la réfraction, celui-ci fonctionne mais est trop bruité (possibilité de le constater en décommentant).

NB2 : L'analyse des matières est proportionnelle à la taille du terrain, mais non le mouvement de l'eau et la camera afin de garder la sensation de grandeur/ou non de celui-ci.

NB3 : Les textures ont été changées pour plus de lisibilité et éviter un effet de « bruit ».

BONUS : Il est possible de visualiser les régions avec la touche « l »

Terrain de 10x10 régions de taille 50 (20-30 us pour l'affichage de toute la map)