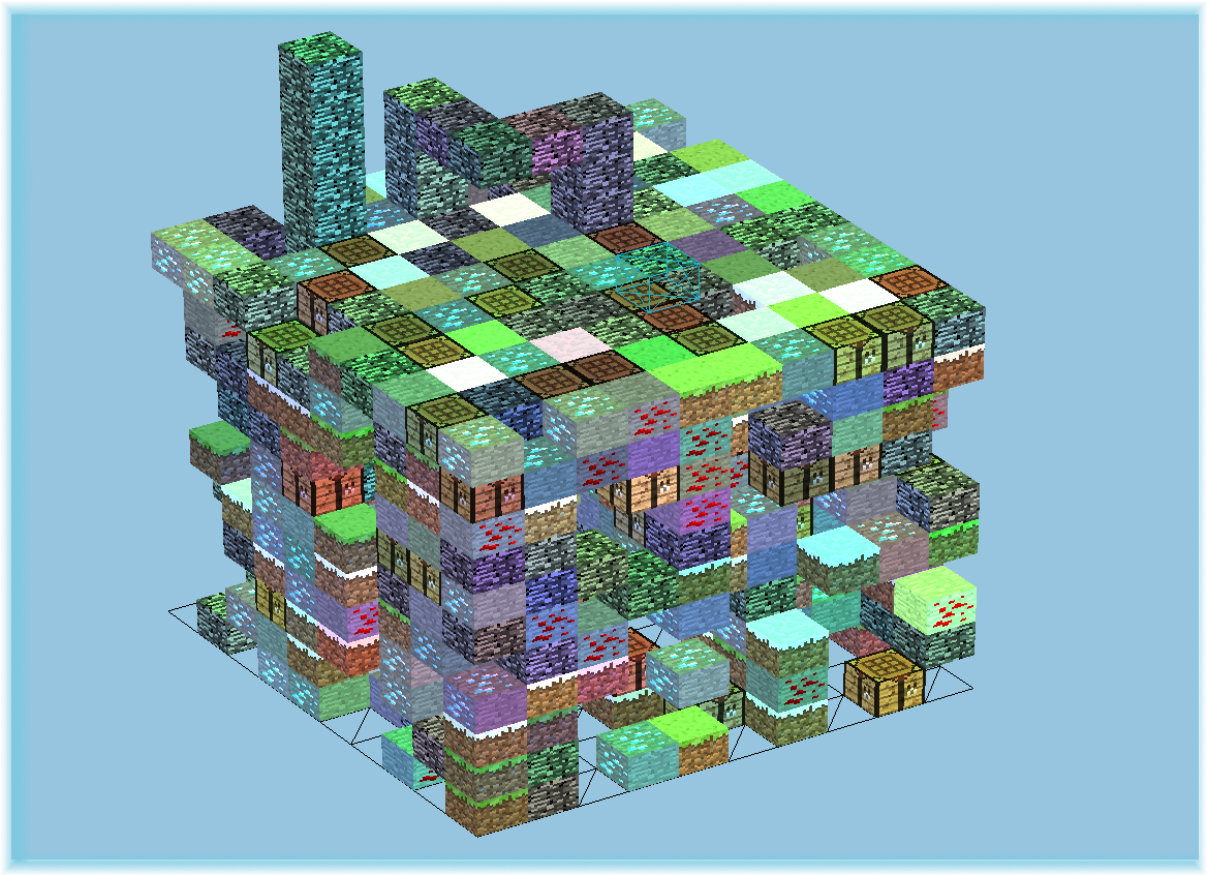
Projet de programmation

World Imaker



KOEPPEL Yoann

THIEL Pierre

Fonctionnalité	Implémentation	Remarques
Affichage d'une scène avec des cubes	**Oui**	
Edition des cubes	**Oui**	Édition non graphique
Sculpture du terrain	**Oui**	
Génération procédurale	**Oui**	Génération dynamique: Génération aléatoire
Ajout de lumières	**Oui**	Les lumières ne peuvent pas être déplacées
Bonus :		
Amélioration de la sélection	**Oui**	
Outils de peinture	**Non**	
Sculpture ++	**Non**	
Sauvegarde	**Non**	
Importer / Exporter	**Non**	
Chargement de modèles 3D	**Non**	
Niveau de discrétisation	**Non**	
Blocs texturés	**Oui**	Utilisation des *CubeMaps*

I. Fonctionnalitées

A. Affichage d'une scène avec des cubes

a) Éléments affichables visibles

Nous définissons les éléments qui seront dessinés par OpenGL comme affichables (‘*Displayable’*), et sont composés :

- d'une matière (‘*Material’*), qui comprend leur couleur de base, et les informations relatives à l'éclairage par Blinn-Phong (Kd, Ks, brillance)

- d'un patron d'affichage (‘*DisplayPattern’*), qui correspond à un maillage complet (position des sommets et normales), chargé au lancement de l'application.

Nous ajoutons à cette structure son ancrage (‘*Anchor’*), c'est-à-dire sa position dans l'espace 3D du rendu, et formons un objet qui pourra effectivement être rendu (‘*Renderable’).*

Trois sortes d'éléments sont rendu, et effectivement visibles, dans la scène :
- les *Cubes*
- le *Curseur*
- la *Sélection*

b) Architecture

L'ensemble de l'affichage est géré par le contrôleur d'affichage *DisplayController* (conception MVC).

- Displayer -

L'affichage est amorcé par le *Displayer,* qui transmet la scène à rendre au *Scene Renderer :* il s'agit d'ajouter des *Displayables* aux files de rendu du *Scene Renderer*, sous forme de *Renderables.*

En résumé, le *Displayer* prend la scène (soit le ‘*Model’),* convertit les éléments *Cube, Curseur,* et *Sélection* qui la compose en *Renderable,* avant de les transmettre.

Notons que les *Cubes* sont accessibles par composition à partir de l'ensemble des Cubes (‘*CubeWorld’),* et la *Sélection* à partir du *Curseur.*

- Scene Renderer - … Expert

Le *Scene Renderer* divise les éléments en files de rendus, suivant leur méthode d'affichage (coloré, texturé, maillage seul, ou caché), puis rend les éléments file par file.

Il a accès aux lumières et caméras, et délègue la mise à jour des matériaux, textures, matrices, et lumières au *Buffer Manager.*

Le rendu d'un élément est un appel au dessin du patron associé par le *Pattern Manager* (expert des patrons).

B. Edition des cubes

a) Outils d'édition

Chaque objet ajouté par l'interface est créé avec des paramètres aléatoires.

Le déplacement des lumières ponctuelles et directionnelles n'a pas été implémenté.

L'édition des cubes se fait uniquement au clavier et à la souris. Ci-dessous une liste des actions possibles avec touches associées :

Cube :

- Ajout : [Inser]
- Suppression : [Suppr]
- Creusement : [!]
- Extrusion : [:]
- Changement texture : [TAB]
- Sélection : [ESPACE]
- Annuler sélection : [ANNULER]
- Déplacement : [ESPACE] + [ENTRÉE]
- Génération aléatoire : [n]

Lumière :

- Ajout ponctuelle : [p]
- Ajout directionnelle : [m]
- Suppression ponctuelle : [9]
- Suppression directionnelle : [0]
- Allumage ambiante : [i]
- Extinction ambiante : [o]

Caméra :

- Rotation : [SOURIS-BOUTON-MOLETTE] + [MOUVEMENT]
- Agrandissement : [SOURIS-MOLETTE]

Le curseur et les outils de génération procédurale seront détaillés plus loin.

b) Architecture

L'interface est supervisée par le contrôleur associé *InterfaceController.*

- Interface -

L'ensemble de l'interface utilisateur est générée par la classe Interface. Elle peut ainsi agir sur le *Model* (ajout de Cubes, Lumières etc.), a accès au curseur, et à la fenêtre SDL à partir du *Displayer* pour la gestion des évènements.

- Cursor -

Le curseur est symbolisé par une position dans l'espace de cube, il connaît *CubeWorld* et est capable de garder en mémoire une *Selection:*(pour de la génération procédurale). Il est en charge de trouver un cube dans *CubeWorld*, de fournir une position pour le creusement et l'action sur le cube, et de modifier ses propriétés (position, texture).

Il est affiché sous forme de maillage.

Les événements SDL sont gérés par nature : événements curseur, souris, clavier.

La bibliothèque ImGUI a été intégrée au projet, mais pas à l'interface. Nous avons néanmoins choisi de garder le code d'implémentation dans le projet, pour une utilisation future.

C. Sculpture du terrain

a) Définitions

Nous définissons le « creusement » d'une pile de cube la suppression du cube le plus haut à partir d'un cube donné.

L'« extrusion » sera l'ajout d'un cube dans la première case libre au dessus d'un cube donné.

b) Architecture

- CubeWorld – … Composite

L'espace de cube est représenté comme une matrice de piles de cubes, dont la largeur est l'axe des abscisses, et la longueur l'axe des côtes – x et z. La hauteur en ordonnée est dynamique : on préférait laisser un degré de liberté à l'utilisateur.

Chaque action sur le monde est déléguée aux piles de cubes *CubeStack* qui le compose.

- CubeStack - … Composite

CubeStack représente une pile de cubes aux étages indicés *CubeFloor. Insérer un cube dans la pile revient à ajouter un CubeFloor* contenant un *Cube, et l'étage d'insertion.*

- CubeFloor - ...

Ajouter au cube l'étage d'insertion dans *CubeFloor* permet de représenter des piles dites « creuses », c'est-à-dire qui admettent des vides. L'hypothèse de piles nous paraissait pertinent car elle nous permettait de créer des creux, et avec cela des grottes, voire des structures flottantes.

- Cube -

Un *Cube* n'est défini finalement que par une position dans l'espace qui lui est renseigné par indirection *via CubeStack et CubeFloor*.

c) Algorithmes

Ajout, suppression (*CubeStack)*:

entrée : un cube, un étage

→ parcours des *CubeFloors* de *CubeStack,* jusqu'à trouver l'étage correspondant.

Creusement (CubeStack):

entrée : néant

→ suppression du cube le plus haut

Extrusion (CubeStack) :

entrée : un étage

→ parcours jusqu'à l'étage, puis parcours jusqu'à une case vide, puis insertion d'un cube identique au cube d'en-dessous.

D. Ajout de lumières

a) Spécificités

Notre implémentation des calculs de lumières ponctuelles et directionnelles fait intervenir des fonctionnalités récentes d'OpenGL (4.3 <) : les *Shader Storage Buffer Objects* (SSBO), qui nous permettent d'envoyer un nombre dynamique de lumières (bien que le nombre maximum soit fixé à l'exécution.)

Nous favorisons également des *Uniform Buffer Objects* aux uniformes pour l'envoi des matrices et des matériaux, car ces objets sont utilisés dans tous les *shaders* et ces UBO fournissent une interface commune plus efficace.

b) Architecture

- *LightManager - … Interface*

C'est une interface de gestion des lumières. Elles sont divisées en deux files : les lumières ponctuelles et directionnelles.

- *Ambiant, Point, Direction Light : Abstract Light - … Abstraction*

Les lumières sont divisées en *AmbiantLight*, *PointLight,* et *DirectionLight.* Elle héritent d'une classe abstraite *AbstractLight* et lui ajoutent une position ou une direction (la lumière d'ambiance n'ajoute rien, mais résout l'abstraction.) Les positions et directions sont en mémoire en coordonnées du monde, et sont converties par multiplication à la matrice *ModelView* lors de leur envoi aux *shaders.* Cette envoie a été dans un premier temps optimisé à l'aide d'un patron Visiteur, mais son implémentation complète n'a pas été effectuée et il a été abandonné.

E. Génération procédurale

a) Spécificités

Le sujet proposait de générer procéduralement la scène à partir d'un fichier de configuration. Nous l'avons implémenté en tant qu'outil d'édition. Le curseur peut sélectionner des *Cubes*, leur attribuer un poids, et les ajouter à sa liste de *Sélection.* Les positions des cubes deviennent les points de contrôles, et ses attributs permettent d'interpoler le matériau du résultat.

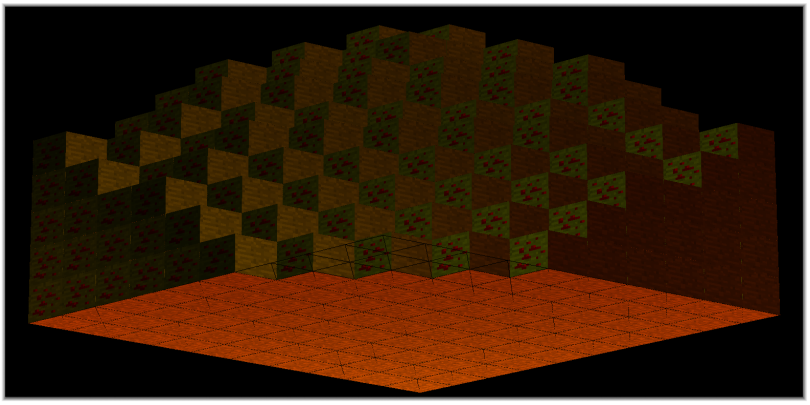
Retour sur l'interface :

- Sélectionner (ajout point de contrôle): [ESPACE]
- Déselectionner : [ANNULER]
- Tout déselectionner : [ÉCHAP]
- Augmenter poids PdC : [↑]
- Diminuer poids PdC : [↓]
- Changer de méthode radiale : [TAB]
- Lancer interpolation : [*]

b) Architecture

- *RadialFunctor, DistanceFunctor - ... Générique, Interface, Stratégie,Usine*

Les fonctions radiales sont nombreuses, et sont toutes utilisées de la même façon : il est donc utile de définir une abstraction *RadialFunctor* qui permet de les appeler de manière générique. Pour aller plus loin dans ce généralisme, nous les implémentons comme des patrons de classes *template*, pour gérer tout type arithmétique. De cette façon, nous pouvons interpoler la couleur, les coefficients de lumière dynamique et statique.



Une scène générée en radial linéaire, avec éclairage directionnel. On observe que les cubes de gauches sont moins brillants que ceux de droites : shininess *croissante.*

La changement d'une méthode à une autre utilise une Stratégie, qui consiste à changer dynamiquement la fonction radiale. Notons que la distance est elle aussi une Stratégie, bien que nous ne nous servons finalement que de la norme L2 : on a cherché une implémentation plus modulaire. Notons également que l'on utilise une *Usine* pour instancier les *RadialFunctor*.

- *RBF, OmegaSolver - ... Générique, expert*

Ces classes effectuent le calcul des *Radial Basis Functions* pour une méthode radiale, une norme, et un ensemble de points de contrôle donnés. *OmegaSolver* est experte en résolution de systèmes linéaires pour la création des omégas.

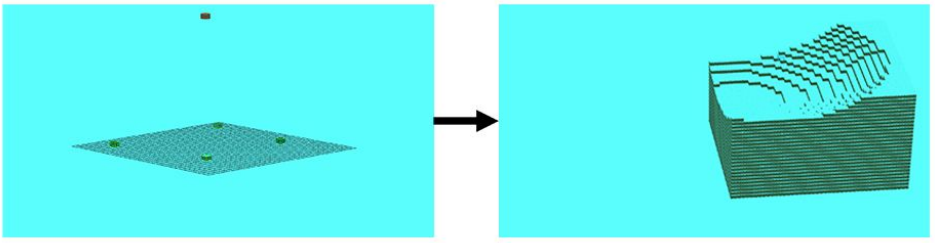
- *ProceduralGenerator - ... Interface*

Cette Classe résout l'abstraction des RBF, en proposant au *CubeWorld* une méthode de création d'une RBF adaptée aux cubes. *CubeWorld* peut ensuite déléguer le calcul du résultat à ses *CubeStacks.*

c) Étude et comparaison des RBF

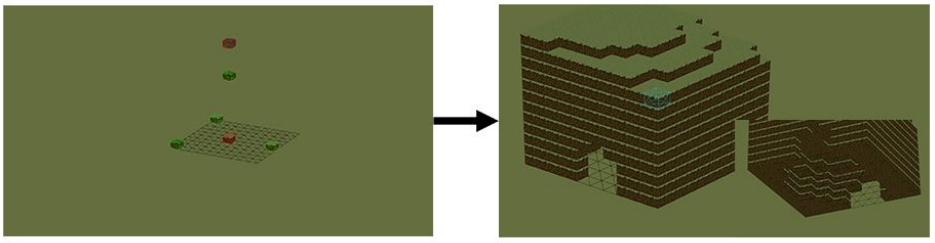
À gauche les points de contrôles, à droite le résultat.

RBF : Linear



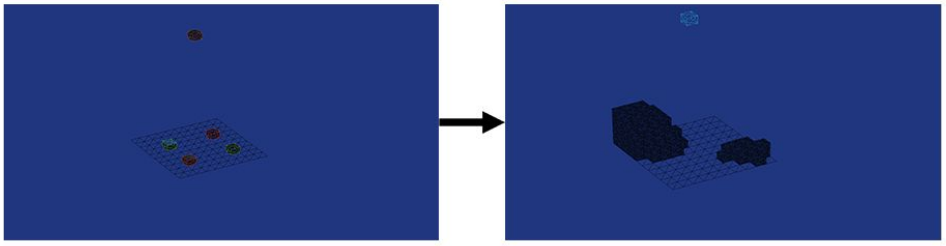
Le rendu de fonction créé des patterns très plat.
Très utile pour représenter des plaines.

RBF : Invert



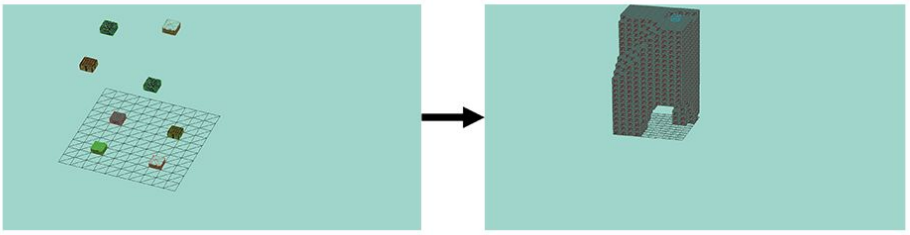
Utilisation dans l'objectif de faire des grottes

RBF : Gaussian

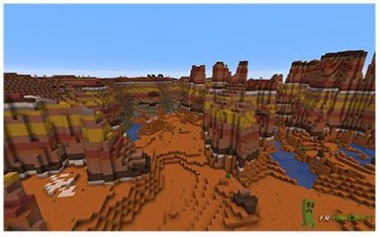


Possibilités de faire des reliefs plus précis, comme des rivières ou des collines

RBF : Multiquadric



Formes droites et strictes, semblable à un cayon



F. Amélioration de la sélection

a) Spécificités

Comme proposé par le sujet, on utilise un *Framebuffer* pour dessiner la position de chaque cube comme une couleur dans une texture.Nous en avons profité pour rendre la scène tout entière dans une texture, avant de l'afficher : il s'agit là d'une méthode efficace pour faire

du post-traitement d'image. Nous n'en faisons pas, mais sommes convaincu que ce prolongement ajoute en modularité.

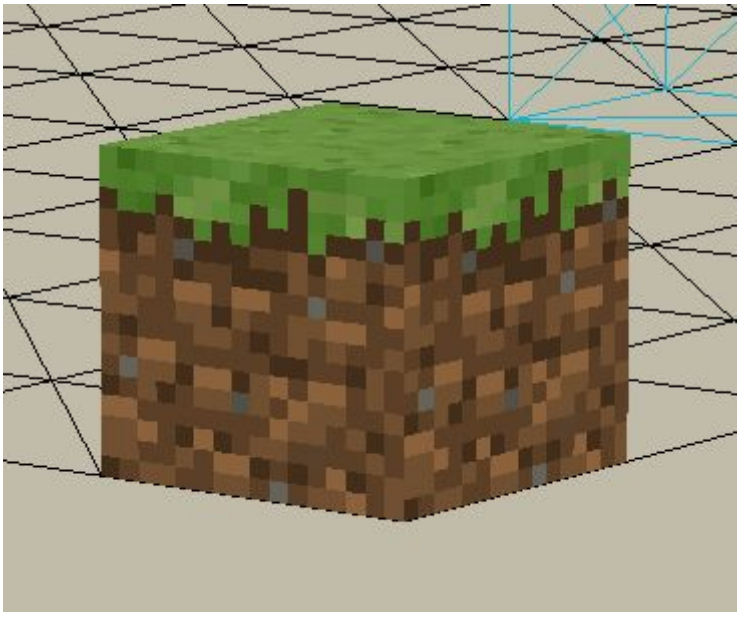
b) Implémentation

Comme dit plus haut, les indices des cubes sont enregistrés dans une texture. L'*Interface* lis les coordonnées de la sélection de souris, pour l'envoyer au curseur qui trouve le cube.

G. Blocs texturés

a) Spécificités

Car on ne dessine que des cubes, nous avons choisi d'utiliser un type de textures spécialisé : les *CubeMaps*. Cela consiste simplement à charger six textures plutôt qu'une seule, et à associer chacune à une face. Cela nous a permis d'afficher des textures du jeu *Minecraft,* qui utilise la même technique.



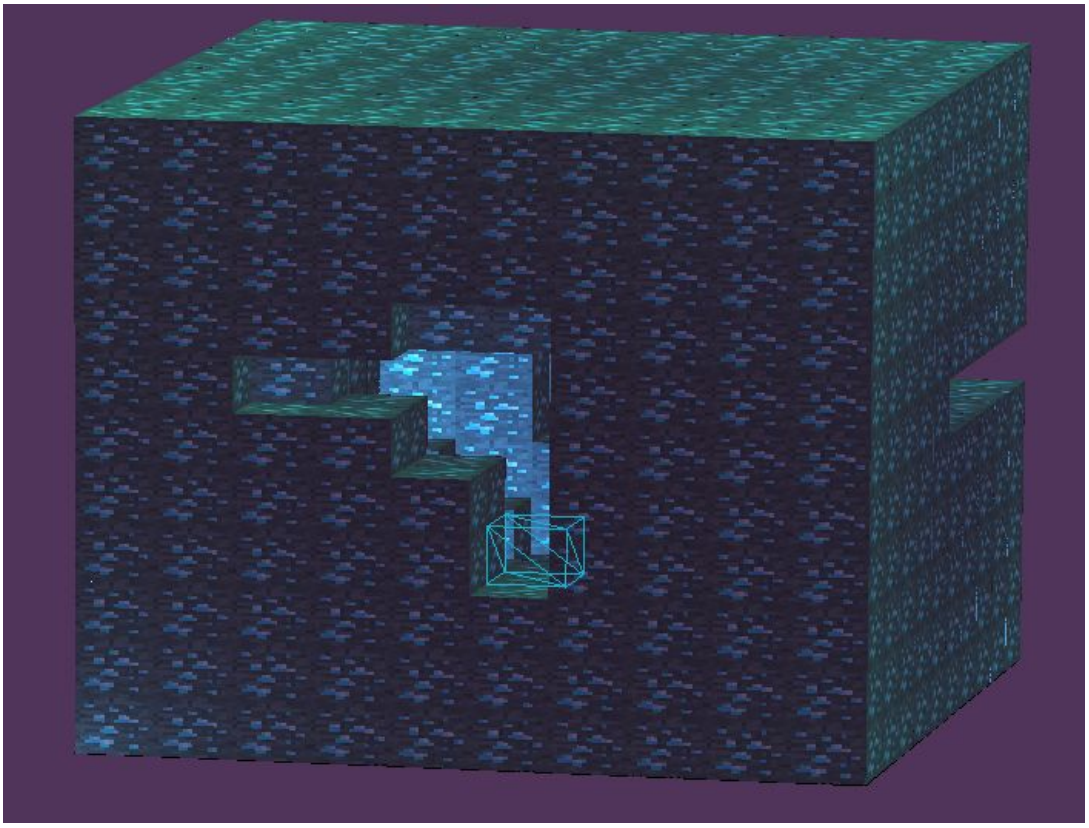
Les textures CubeMap permettent d'afficher des cubes texturés aux faces différents.

b) Architexture

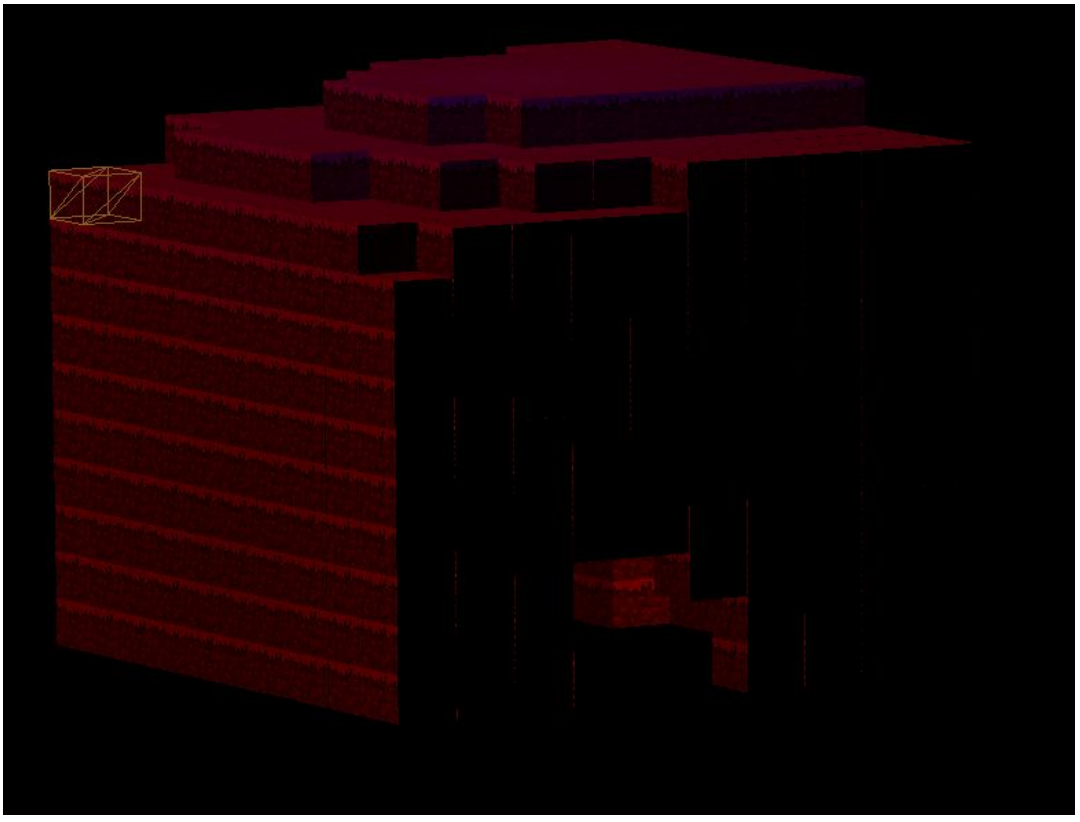
- *TextureManager* - ... Interface

Une fois les textures 2D implémentées, les *CubeMaps* ne nécessite pas de précautions supplémentaires. Nous avons choisi de les charger à partir d'un fichier de configuration. Cette responsabilité incombe au *TextureManager.* Il charge le fichier de configuration dans *resources/textures* dans lequel est donné, pour chaque cube, le nom de la texture de chaque face.

II. Résultats : captures d'écran



Une mine de diamants ? Générée avec RBF linéaire puis creusée, puis lumières ponctuelles.



III. Les difficultés rencontrées

1) Linux et Windows

L'une de nos première ambition était de réussir à faire du multi-plateforme (Linux et Windows) en s'appuyant sur l'environnement de développement CLion. La configuration de l'IDE nous a posé pas mal de problèmes. Nous n'avions pas les mêmes compilateurs entre les deux plateformes, et c'est qu'après plusieurs essaies que nous avons réussi à trouver une solution stable. L'autre grosse difficulté fut la gestion des bibliothèques, traiter différemment entre Linux et Windows. Nous avons réglé le problème en nous plongeons plus profondément dans l'utilisation et la configuration du CMake.

2) ImGUI notre pire cauchemar

Très peu de temps après le début de la programmation de notre projet, nous avons voulu intégrer la librairie ImGUI. Nous espérions, en implémentant très tôt dans le programme maximiser le code et la rendre compatible le plus tôt possible. Nous avons rapidement constaté notre erreur : la librairie nous demander d'implémenter une nouvelle fonctionnalité à tel point que nous avons finis par nous dire que nous allions achever le projet avant de réussir à la faire fonctionner. On a donc dû très rapidement la mettre de côté puisqu'elle nous empêchait de compiler et vérifier le fonctionnement du programme.

3) Shaders, GLSL

Nous nous sommes également rendu compte que l'optimisation du code par le compilateur n'était pas forcément notre meilleur ami. Parfois, il supprimait des variables qu'il jugeait inutiles, notamment dans les shaders. De plus, l'erreur affichée était que le compilateur ne trouvait pas la variable. Comme la variable existait pour nous et qu'elle était au bon endroit, nous pensions que le problème venait de la portée. C'est seulement après avoir perdu beaucoup de temps que nous avons compris d'où venait l'erreur.

L'utilisation des SSBO et UBO a été dans un premier temps dûr à mettre en place, car nous n'avions pas vu ces fonctionnalités en TD. En particulier, on citera le standard de stockage des variables uniformes '*std140*' d'OpenGL qui a nécessité des ajustements dans nos structures de données

4) Modernité

L'utilisation de propriétés modernes telles que la norme c++17 empêche la compilation sur les ordinateurs de la faculté (en c++14). L'unique blocage est un *if constexpr* utilisé dans la génération aléatoire, et ne peut pas être contourné simplement (avant c++11, on aurait utilisé *static_if*). Deuxièmement, les SSBO n'étant apparu que depuis OpenGL 4.3, il nous a parfois été difficile d'exécuter le programme sur des machines dont les pilotes sont obsolètes.

IV. Retour individuel

Pierre :

Ce projet était l'occasion de confirmer mes acquis sur la programmation orientée objet et la synthèse d'image moderne. J'ai cependant trop rapidement voulu aller trop dans le détail, et trop compliqué : en clair, je n'ai pas assez pris en compte les difficultés de Yoann. En définitive, il a été contraint de décrocher de ce que je faisais, et j'estime qu'il aurait appris beaucoup plus si l'on était allé à l'essentiel. J'ai ignoré le fait qu'il s'agissait aussi d'une occasion de travailler en équipe. Malgré tout, mon expérience de travail avec Yoann est très positive et encourageante, et j'ai appris beaucoup lors de notre travail en commun.

Yoann :

Je suis très partagé vis-à-vis de ce projet. D'un côté, l'expérience de travail avec Pierre était extrêmement enrichissante. D'un autre côté, j'ai l'impression de ne pas avoir été d'une grande aide dans le projet. Le travail est complètement déséquilibré. Pierre a pu m'apporter beaucoup de connaissances, répondre à mes questions et me donner des explications détaillées. Grâce à Pierre, j'ai pu apprendre beaucoup de choses. De mon côté, j'ai eu beaucoup de mal à produire quelque chose d'assez bon pour que Pierre n'ait pas à le corriger derrière. J'espère pouvoir retravailler avec Pierre sur un autre projet où je pourrais lui apporter plus de choses.

V. Appendices (Annexes ?)

1. Diagramme de classe simplifié (UML) :

Certaines classes gérées par les *'Managers'* ont été omises car moins représentatives de la structure générale.