BARBARA GUYONNEAU - ANDRÉA GUILLOT

WORLD IMAKER

EDITEUR VISUALISATEUR DE TERRAIN

https://github.com/AndreaGuillot/World_IMaker

1. CAHIER DES CHARGES

	Détails	Etat
AFFICHAGE SCÈNE AVEC CUBES	▪ Scène initiale	Fait
	▪ Moteur de rendu + Shaders	Fait
EDITION DES CUBES	▪ Déplacer le curseur avec le clavier	Fait
	▪ Dessiner les contours du curseur	Fait
	▪ Curseur toujours visible	Fait
	▪ Modifier couleur du cube sélectionné	Fait
SCULPTURE DU TERRAIN	▪ Ajouter, supprimer, extruder, creuser un cube	Fait
GÉNÉRATION PROCÉDURALE	▪ Générer une scène avec les RBF	Fait
LUMIÈRES	▪ Changer de mode	Pas fait
	▪ Lumière directionnelle	Fait
	▪ Point de lumière	Fait
	▪ Positionner ces objets	Pas fait
SAUVEGARDE / CHARGEMENT	▪ Sauvegarde dans un fichier	Fait
	▪ Lecture du fichier	Fait
	▪ Options disponibles dans les menus	Fait
	▪ Chemin spécifié par l'utilisateur	Fait
BLOCS TEXTURÉS	▪ Définition de sets de textures	Fait
	▪ Ajout de texture	Ne fonctionne pas : la texture s'applique à l'ensemble des cubes.

Additionnel	Détails	Etat
AUTRES	▪ Réinitialiser la scène de départ	Fait
	▪ Supprimer tous les cubes	Fonctionne partiellement : ne supprime que les cubes contenus dans le volume “world”.

2. DESCRIPTION GLOBALE DE L’ARCHITECTURE DE NOTRE PROGRAMME

lib	include	src	shaders	assets	doc	CMakeLists.txt

LIB

Contient les librairies utilisées pour le projet :
- **glimac et glm :** Ces deux librairies ont été importées depuis nos TD de synthèse de l’image.
La lib *glm* nous permet notamment de manipuler des matrices et des vecteurs. Quant à la lib *glimac*, elle possède des classes déjà prêtes qui simplifient l’initialisation de certaines structures et/ou fonctionnalités.
- **imgui :** Cette librairie (https://github.com/ocornut/imgui) nous fournit les outils pour créer différents types de menus.

INCLUDE

Contient les prototypes des fonctions et les structures (.hpp) :
- **Cubes :** regroupe les attributs permettant de créer un cube et gère également l’ensemble des cubes créés.
- **Curseur :** classe fille de Cubes. Elle gère les positions du curseur.
- **GameControls :** permet l’affichage des commandes utilisateur dans le terminal.
- **Interface :** permet d’initialiser la fenêtre de OpenGL ainsi que les menus ImGui.
- **Map :** génère des scènes et ajoute/supprime des points de control pour la génération procédurale.
- **RadialBasisFunctions :** calcul les radial basis functions pour la génération procédurale.
- **ShaderProgram :** gère le chargement des shaders des différents objets créés.
- **Texture :** initialise une texture.
- **TrackballCamera :** créer une caméra qui tourne autour de l’origine du repère

SRC

Contient les fonctions (.cpp) :
Le découpement est le même que dans le dossier include. On y trouve en plus le fichier *main* qui appelle les fonctions créées.

SHADERS

Contient le vertex shader, commun au cube et au curseur, ainsi que le fragment shader du cube et le fragment shader du curseur.

ASSETS

Contient les textures des cubes au format png ou jpg.

DOC

Contient le sujet de ce projet ainsi que son rapport, que vous lisez en ce moment.

CMAKELISTS.TXT

Permet la compilation des fichiers.

3. DESCRIPTION DES FONCTIONNALITÉS

3.1 Affichage d'une scène avec des cubes

▪ Créer une scène initiale :

Notre monde est défini par une constante *world_taille* qui limite sa hauteur, sa longueur et sa profondeur. Pour créer **un** cube, nous nous sommes inspirées de la classe Sphère de la lib glimac.

Cubes
m_vertex vbo vao ibo iboWireframe
Cubes() ~Cubes() drawCube()

Ainsi, pour définir un cube, nous utilisons un std::vector de type *ShapeVertex* qui stocke les informations sur les coordonnées des sommets, des normales ainsi que des textures d'un cube. Ces informations sont initialisées dans le constructeur.
Au même endroit vient ensuite l'initialisation des buffers. Par ailleurs, nous utilisons des buffers instanciés afin d'avoir un seul exemplaire de chaque sommet. L'un deux nous permettra par la suite de dessiner seulement les contours "cubiques" du curseur.

Dans le destructeur, nous avons implémenté les fonctions permettant de libérer de la mémoire (glDeleteBuffers, glDeleteVertexArrays).

L'appel à la fonction drawCall permet finalement le dessin du cube.

Afin de dessiner plusieurs cubes, nous créons un buffer de plus contenant leur position. Nous les stockons ensuite dans un std::vector. Nous réalisons la même opération afin de stocker leur couleur. Chaque fois qu'un nouveau cube est créer ou supprimer (à partir de sa position), une nouvelle liste de cubes est envoyée à OpenGL via updateGPU.

Cubes
m_position m_color vbPos vbCol
findCube() addCube() removeCube() updateGPU()

Map
...
initWorld()

Au commencement, l'utilisateur voit une scène de 3 couches de cubes de couleurs différentes représentant le fond du monde. (La fonction initWolrd prend en paramètre un objet de type Cubes).

▪ Visualiser la scène :
Afin de visualiser la scène, nous avons repris la caméra TrackballCamera vue en TD. L'utilisateur peut la manipuler via le clavier (*moveFront, moveUp, moveLeft*) et la souris (*rotateLeft*).

3.2 Édition des cubes

▪ Curseur :

Le curseur est une classe fille de Cubes. C'est lui qui permet à l'utilisateur de "sélectionner" un cube. Nous le rendons toujours visible en utilisant la fonction glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) qui active la transparence des cubes.

Curseur
m_posCursor m_colorCursor
drawCurseur() deplacement() onKeyPressed() getPosition()

Tout comme un cube "classique", le curseur possède une position ainsi qu'une couleur (blanc). La dessin du curseur est différent de celui d'un cube. Seuls ses contours sont représentés.

A chaque déplacement du curseur, via les fonctions onKeyPressed et deplacement, le curseur à la position p-1 est supprimé et un nouveau est dessiné à la position p.

L'édition des cubes est ensuite possible grâce au getPosition du curseur qui renvoie la position du curseur et ainsi, du cube sur lequel il se trouve.

▪ Couleurs :

Nous avons choisi trois couleurs possibles pour un cube (rouge, vert, bleu). Via la position du curseur, il est possible de changer la couleur d'un cube : cube.**editColor(**cursor.getPosition(), color**)**
Grâce à la position p, la fonction parcourt le std::vector qui stocke les couleurs et remplace la valeur initiale située à l'index p par la nouvelle valeur de couleur entrée en paramètre.
Cette fonctionnalité est disponible via le menu ImGui.

3.3 Sculpture du terrain

▪ Outils de sculpture :

Cubes
...
isCubeExist() extrudeCube() digCube()

Les fonctions addCube, removeCube, extrudeCube et digCube permettent de sculpter le terrain. Un cube ne peut pas être construit par dessus un autre.

- addCube dessine un cube avec la couleur du dernier cube dessiné.
- removeCube supprime le cube sélectionné.
- extrudeCube ajoute des cubes au sommet de la colonne, ils sont de la même couleur que celui sur lequel se trouve le curseur.
- digCube supprime les cubes en partant du haut de la colonne.

Ces fonctionnalités sont disponibles via le clavier ou via un menu ImGui.

3.4 Génération procédurale

▪ La map :

L'utilisateur peut activer ou désactiver la fonction "génération procédurale" et peut aussi ajouter et supprimer des points de contrôle. Et tout ceci grâce à Map.cpp/hpp, où l'on définit la classe map qui a pour attributs :

- Une liste de points de contrôle : soit une MatrixXf ControlPoints(nbDePoints, 2).
- Une liste de valeurs associées une à une à ces points : soit un VectorXf(nbDePoints)

- Notre map : soit une MatrixXd(taille, taille) contenant les hauteurs de tous les points de notre carte

et pour méthodes, inspirées par celles de la classe Cubes :

- addControlPoint
- removeControlPoint
- findControlPoint
- loadWorld parcourt le monde, ajoute un cube suivant notre MatrixXd map et fait le lien avec RadialBasisFunction.hpp/cpp.
- ClearWorld supprime les cubes liés à la génération procédurale de la liste des cubes à dessiner. Les points de contrôle sont toujours en mémoire.

▪ Le calcul :

D'après notre cours de math, pour calculer nos points on utilise la formule suivante :

$$g(x) = \sum_{i=1}^{k} w_i * \Phi(\left| x - x_i \right|)$$

Pour cela nous avons utilisé Eigen.
Dans RadialBasisFunctions.cpp/hpp nous avons donc définie :

- findOmega qui calcule puis renvoie un VectorXf contenant nos ω.
- findValue qui renvoie la valeur de Φ.
- getMap qui remplie notre MatrixXd map.
- phi, phi1, phi2 les fonctions radiales (cf 4.)

Et bien sûr nous avons modifié interface.cpp afin d'afficher le menu Imgui lié à la génération procédurale et d'afficher certaines informations dans le terminal comme les différents points de contrôle ou d'éventuelles erreurs.

3.5 Lumières

La première étape pour les lumières a été de créer un tableau de vecteurs contenants les normales de notre cube (dans Cubes.cpp), ceci permettant de traiter chaque face du cube indépendamment des autres. Les normales indiquent en effet la direction de chaque face et permettent de calculer si notre face reçoit plus ou moins de lumière en fonction de son inclinaison par rapport à la source lumineuse.
Les calculs de lumières ont, eux, été fait dans le fragment shader du cube.

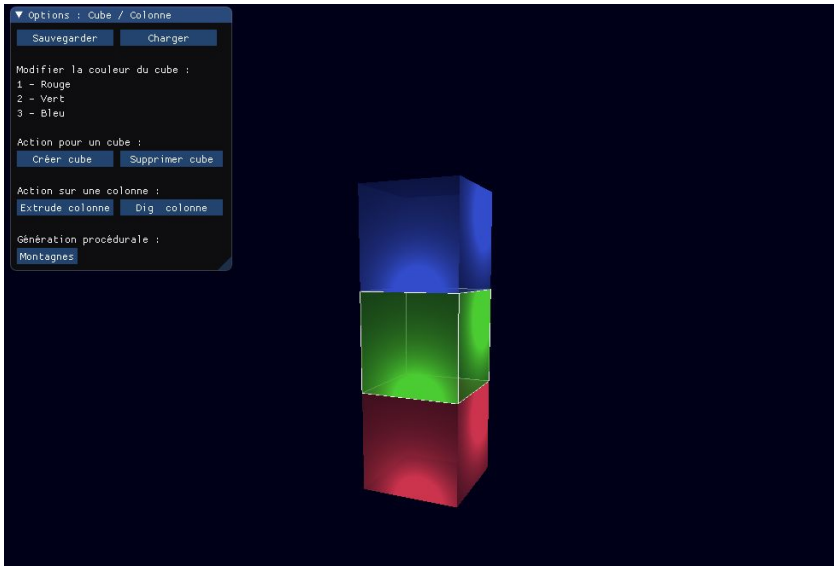
▪ Lumière directionnelle :

La lumière directionnelle de type soleil est créée à l'aide d'un produit scalaire entre ces normales et la direction de la lumière. On mesure ainsi à quel point la lumière arrive perpendiculairement à la face.
Puis on fait un maximum avec 0.2 pour éviter d'avoir une luminosité négative ou trop faible.
On a alors : `float luminosityDirLight = max(-dot(vNormal, direction), 0.2);`

Puis on multiplie cette luminosité par la couleur de notre cube :
`fFragColor = vec4(vColor * luminosityDirLight);`

- Lumière ponctuelle :

Pour la lumière ponctuelle c'est la même chose sauf qu'on donne la position de la lumière pour ensuite déterminer la distance entre elle et notre pixel.
Différentes lumières peuvent être additionnées.



3.6 Scène

- Sauvegarder et charger une scène :

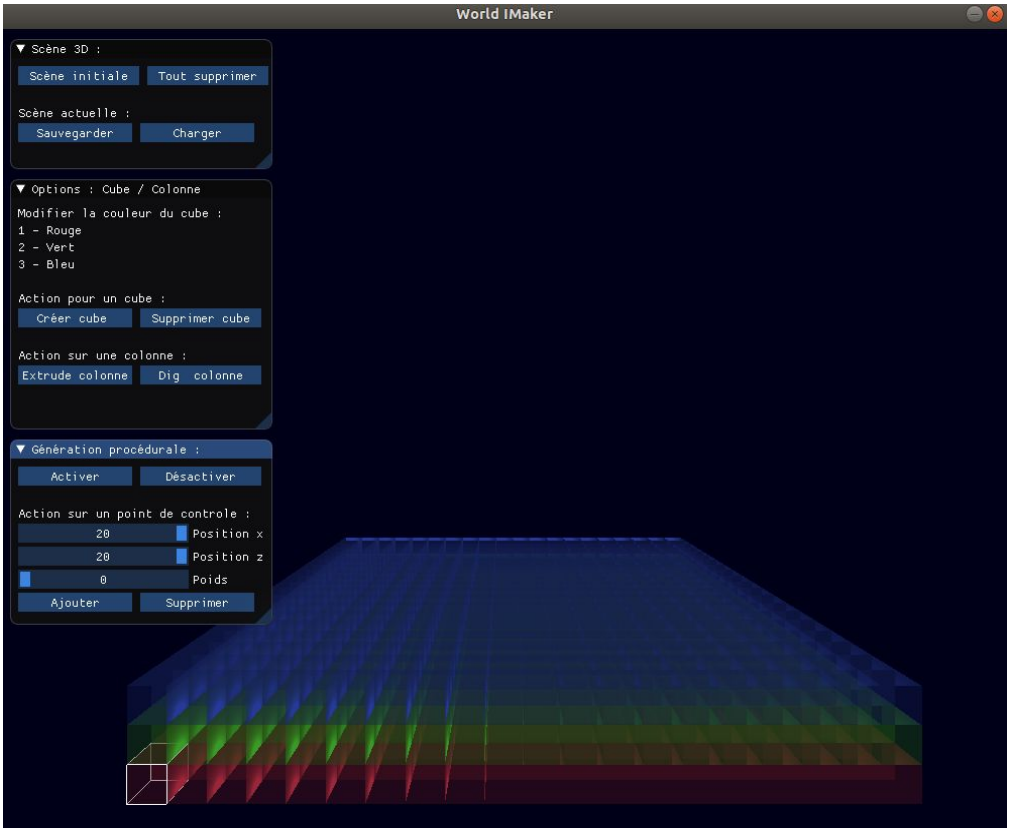
Les fonctions saveScene et loadScene définies dans Cubes permettent de stocker les différentes informations dans un fichier .txt puis de les lire.
Ainsi, avec std::ofstream, le fichier stocke les valeurs de position, de couleur (et de texture) de chacun des cubes. Avec std::ifstream, il est ensuite possible de restituer ces valeurs.
Ces deux fonctionnalités sont disponibles via l'interface ImGui. Lorsque l'utilisateur sélectionne l'une d'elles, via le terminal, il lui est demandé d'entrer le nom du fichier ainsi que son chemin.

■ Recréer la scène initiale :

Pour que l'utilisateur puisse repartir de la scène initiale s'il le souhaite, nous avons rendu disponible la fonction initWorld de la classe Map via l'interface ImGui.

■ Supprimer la scène :

Pour que l'utilisateur puisse tout supprimer et repartir de zéro, nous avons rendu disponible, via ImGui, une fonction qui supprime tous les cubes présent dans notre monde.



3.7 Blocs texturés

■ Texture :

Texture
m_path m_textureID imgPointer
Texture() ~Texture() bindTexture() unbindTexture() setTexturePath()

Pour créer une texture, nous avons repris la méthode vue en TD en incluant donc le header Image qui permet de charger une image.

Dans le constructeur, la texture est créée. Il prend en paramètre le nom de la texture ce qui permet de trouver son chemin (grâce à setTexturePath) et donc de l'initialiser. Dans le destructeur, nous faisons appel à la fonction glDeleteTextures pour libérer de l'espace.

Les fonctions bindTexture et unbindTexture permettent comme leur nom l'indique de binder et débinder une texture. Elles sont utilisées dans la fonction de dessin de Cubes.

▪ L'implémenter :

Cubes
Texture* texture m_texture m_type
editTexture() addTexture()

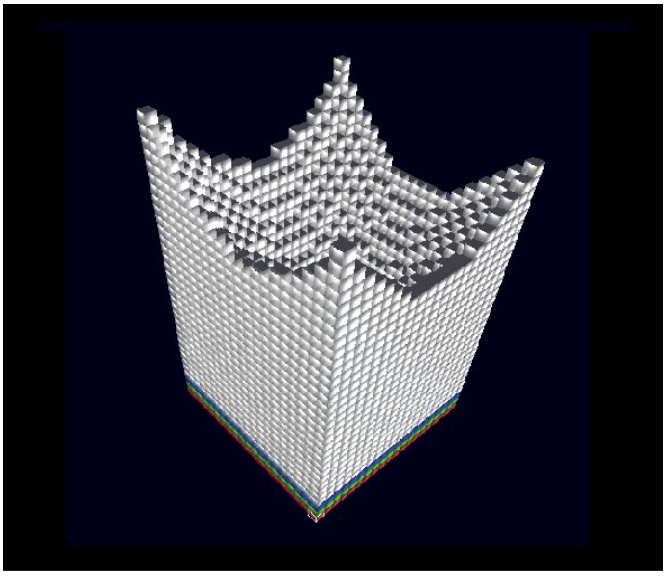
Chaque texture correspond à un entier (m_type) afin de faciliter son utilisation dans d'autres fonctions (ex : sauvegarder scène). Tout comme pour la position ou la couleur, nous stockons ces valeurs dans un std::vector (m_texture).

- La texture est initialisée dans le constructeur de Cubes.
- editTexture fonctionne comme editColor : grâce à la position p du curseur, la fonction parcourt m_texture et remplace la valeur initiale située à l'index p par la nouvelle valeur.
- addTexture renvoie à partir du m_type du cube la texture correspondante.

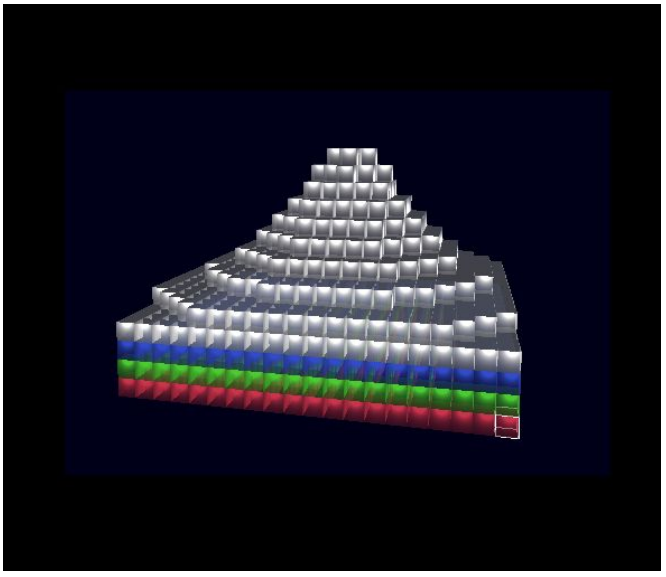
4. LES FONCTIONS RADIALES : BILAN DE NOS OBSERVATIONS

Chaque fonction a été testé sur un point de contrôle en (10,10) avec un poids de 10.

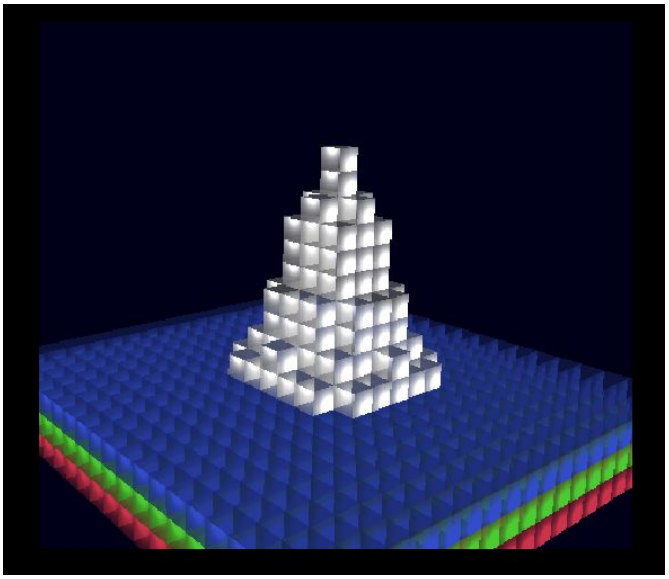
▪ Multiquadrique $\varphi(d) = \sqrt{1 + (\varepsilon d)^2}$ (phi2) :



▪ Inverse quadratique $\varphi(d) = \frac{1}{1+(\varepsilon d)^2}$ (phi1) :



▪ Gaussienne $\varphi(d) = e^{-\varepsilon d^2}$ (phi) :



Nous avons choisi de garder la fonction gaussienne car elle ressemblait plus à ce que l'on souhaitait faire : des montagnes. De plus, nous avons un monde de 20 par 20 (pour faciliter l'exécution du programme sur nos ordinateurs peu puissants), la génération procédurale via la fonction gaussienne est donc plus adaptée puisqu'elle prend moins de place. On peut s'amuser à faire plein de montagnes !

5. DIFFICULTÉS

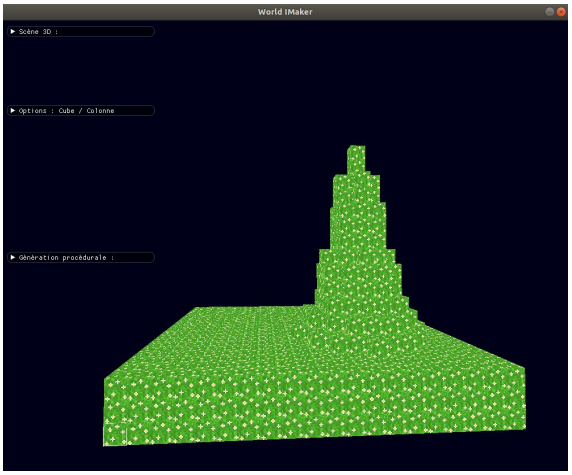
▪ **Cube** :

La première difficulté que nous avons rencontrée a été de créer un cube. Nous avions déterminé qu'il s'agirait de la première étape du projet mais nous ne savions pas du tout par où commencer. C'est en se plongeant de manière plus poussée dans les TD que nous avons trouvé une piste. En effet, nous avions vu comment créer une sphère en 3D. Nous avons donc trouvé plus facile de partir de cette structure et de la simplifiée afin d'obtenir un cube. Il a donc fallu s'assurer de bien comprendre cette structure préexistante afin de l'utiliser correctement et de ne pas supprimer d'éléments importants.

▪ **ImGui** :

Prendre en main ImGui nous a pris beaucoup de temps. En effet, cette librairie propose une énorme quantité de fonctionnalités. Les premières fois où nous nous sommes plongées dedans, il était assez compliqué de trouver la fonctionnalité qui correspondait le mieux à nos besoins.

▪ **Texture** :



Le problème rencontré (et non-résolu) est qu'une texture s'applique à tous les cubes. Nous avons supposé que le problème venait du slot de la texture. En effet, nous pensons que les textures créées s'attache toujours au même endroit.
Piste envisagée : intégrer cette nouvelle composante en utilisant **glActiveTexture(**GL_TEXTURE0 + m_textureSlot**)** afin d'envoyer des valeurs différentes à notre uniform sampler2D mais cela n'a rien changé.
Nous avons également pensé que tout comme pour les

positions et les couleurs, il faudrait créer un nouveau buffer qui stocke le slot de texture. que le cube doit utiliser. Nous avons cependant rencontré des erreurs de segmentation dont nous n'avons pas trouvé l'origine et qui nous ont obligé à laisser de côté cette idée.

- **Supprimer la scène** :

Pour supprimer la totalité des cubes de la scène, nous avons dans un premier temps pensé à utiliser la fonction clear() et ainsi vider les vectors contenant les positions, les couleurs (et les textures). Nous n'avons pas réussi cela sans nous l'expliquer. Les vectors restent "intactes". Nous avons donc opté pour une autre méthode en utilisant la fonction remove (de Cubes). Toutefois, cette dernière a ses limites puisque nous avons dû indiquer le volume dans lequel se trouvent les cubes à supprimer. Ainsi, si un cube est créé à l'extérieur de ce volume, il ne sera pas supprimé. Piste envisagée : Ajouter des contraintes dans la fonction add (de Cubes) afin de limiter la création de cubes au volume de notre monde.

- **Lumières** :

Par la suite, nous aurions souhaité permettre à l'utilisateur d'ajouter différentes lumières ponctuelles et/ou directionnelles ainsi que lui permettre de choisir entre le mode jour ou le mode nuit. En effet, pour le moment tout doit être modifié dans le code. Nous avons malheureusement manqué de temps et eu quelques soucis avec nos variables uniformes qui ne se modifiaient pas.

- **Génération procédurale** :

Si vous nous aviez vu lorsque nous avons lu dans le sujet les mots "génération procédurale" et "fonctions radiales", vous auriez pu voir la panique dans nos yeux.
Bien comprendre les notions de math et comment mettre en place ces radial basis functions n'était pas une tâche facile. L'aide de nos camarades comme Jules Fouchy, Margaux Vaillant ou encore Zoé Durand n'était pas de trop. Après qu'ils nous aient répété 5 fois comment ça fonctionnait, petit à petit le chemin s'est dessiné et nous avons pu continuer seules notre route.

6. GUIDE UTILISATEUR

Instructions commandes :

1) Créer un dossier build : $ mkdir World_IMaker_build
2) Se placer dans ce dossier : $ cd World_IMaker_build
3) Lancer le cmake : $ cmake ../World_IMaker
4) Lancer le make : $ make
5) Exécuter le programme : $ ./bin/WorldIMaker

Guide utilisateur :

Commandes clavier :

Mouvement caméra : GAUCHE : **W** DROITE : **X**	**Mouvement curseur** : GAUCHE : **Flèche gauche** DROITE : **Flèche droite**	**Action cube :** Ajouter : **Barre espace** Ajouter rouge : **C**

HAUT : **A** BAS : **Q** AVANCER : **Z** RECULER : **S** ROTATION : **Souris**	HAUT : **Flèche haut** BAS : **Flèche bas** AVANCER : **P** RECULER : **M**	Ajouter vert : **V** Ajouter bleu : **B** Supprimer : **Suppr** Extrude : **E** Dig : **D**

Interface ImGui :

- Scène : recréer scène initiale, tout supprimer, sauvegarder scène actuelle, charger une scène
- Options sur les cubes : changer couleur, add, remove, extrude, dig
- Génération procédurale : activer/désactiver, ajouter/ supprimer (position, poids)

7. CONCLUSION

Andréa

N'étant pas une programmeuse née, j'appréhendais beaucoup ce projet qui me paraissait colossal. Bien que ce projet m'ait pris beaucoup de temps en matière de réflexion et de débogage, c'était très satisfaisant de pouvoir appliquer ses connaissances dans un environnement concret.
J'ai trouvé que les TD de synthèse de l'image apportent de bonnes bases pour pouvoir commencer le programme. Le continuer était une autre affaire. Toutefois j'ai réellement l'impression d'avoir pu progresser que ce soit en analyse ou bien en codage pur et brute. J'ai également apprécié ce travail en binôme qui a été efficace que ce soit en répartition du temps ou du travail.

Barbara

Habituellement, je panique devant un projet de cet envergure, mais les TDs de synthèse de l'image étant vraiment clairs et détaillés, j'ai pu aborder ce projet beaucoup plus sereinement que tous les précédents. Evidemment ça ne veut pas dire que je voyais comment réaliser le projet ni que ça me semblait simple, bien au contraire, mais au moins que je voyais par où commencer.
Après réalisation de ce projet, je suis plutôt satisfaite de ce que nous avons réussi à produire malgré une grosse déception au niveau du mode jour/nuit que je n'ai pas eu le temps de finir. La bonne entente et la répartition des tâches au sein du binôme nous ont permis de mener à bien ce projet. C'était un vrai plaisir de travailler avec Andréa et de constater l'entraide toujours présente à l'IMAC. Je pense avoir acquis de nombreuses connaissances particulièrement en architecture logicielle et analyse. Cependant la gestion du temps reste un point à améliorer de mon côté.

Nous espérons que Toto sera embauché suite à son entretien.