WorldIMaker

Laurine Lafontaine - Flora Mallet - IMAC 2

OpenGl - Maths

2020

INTRODUCTION

Notre projet consiste en la réalisation d'un éditeur-visualiseur de terrain et de scène en 3D. Le but d'un éditeur-visualiseur est de proposer à l'utilisateur un programme : l'utilisateur est capable de créer un monde à partir de blocs cubiques et de naviguer dedans. Ce projet a pour but de consolider nos connaissances en programmation C++, du rendu 3D et de l'interpolation numérique.

Afin que le projet fonctionne correctement, vous aurez besoin d'installer le package GLUT, OpenGL, CMAKE, GLEW, SDL2, SDL2 Mixer et Doxygen. Tout est très bien expliqué dans le README du repo github !
(https://github.com/LafLaurine/imac2-worldIMaker/blob/master/README.md)

PRÉSENTATION PROJET

Amusez-vous à créer votre monde rempli de petits cubes de couleurs différentes, grâce à notre éditeur-visualiseur. Au revoir les logiciels Blender, Maya, 3DSmax et à bien d'autres encore et dites bonjour à WorldIMaker.

Notre éditeur-visualisateur s'inspire du principe du mode de création dans Minecraft. Ainsi, vous retrouvez des textures du dit jeu ainsi que des sons tirés de celui-ci (copyright Minecraft bien évidemment).

Exécution de l'application :

Faites un git clone du projet : https://github.com/LafLaurine/imac2-worldIMaker
Créez un dossier build. Allez dans ce dossier et faites la commande cmake.. puis make. Ensuite, afin de compiler le fichier Doxyfile html, il va falloir vous rendre dans le dossier doc du build et taper la commande make html.
Vous avez maintenant accès au fichier Doxygen en allant dans le dossier doc-doxygen puis html et enfin, ouvez le fichier index.html.

Encore une fois, l'exécution est bien expliquée dans le README.

Si l'envie vous prend, vous pouvez tenter de créer votre propre fichier de point de contrôle de radial basis function : modifiez un des nôtres dans ./assets/doc ou bien créez en un nouveau (attention, s'en suit une manipulation à faire dans le main.cpp ou vous devez ajouter à la main le nouveau fichier créé). Vous pourrez ainsi probablement tester notre jeu avec vos propres conditions !

Au niveau du choix graphique, on commence avec une scène rempli de cubes blanc pour représenter la scène. Ce choix a été fait pour combler notre manque de neige en cet hiver de réchauffement climatique.

ARCHITECTURE DU PROJET

imac2-worldIMaker

Structure architecturale :

assets/

doc/

contient les fichiers de points de contrôle des RBF

sounds/

contient les sons de notre programme

textures/

contient les textures de notre programme

build/

contient les fichiers générés par le cmake, utiles à la compilation

CMake/

FindSDL2.cmake

FIndSDL2_MIXER.cmake

doc/

contient les fichiers utiles à Doxygen :

CMakeLists.txt

Doxyfile-html.cmake

lib/

glimac/

librairie créée par l'IMAC. Contient les fichiers principaux de notre application, qui découlent du namespace glimac

CMakeLists.txt

include/

.hpp de nos fichiers

src/

.cpp de nos fichiers

imgui/

librairie de gestion d'interface graphique

CMakeLists.txt

include/

src/

SDL2-2.0.10/

fichiers sources de SDL2 pour Windows.

include/

lib/

shaders/

shaders de notre application

colorCube.fs.glsl

colorCube.vs.glsl

texture.fs.glsl

texture.vs.glsl

src/
fichier principal du projet (+ sound car pas la possibilité de le faire fonctionner autre part)
main.cpp
sound.hpp
sound.cpp

third-party/
autres librairies
include/glm

CmakeLists.txt
License
Readme.md

Structure logicielle du projet :

dans le dossier src
appartient à glimac

main.cpp	Le fichier main.cpp rassemble les fonctions des structures ci-dessous en un programme fonctionnel.
Sound.cpp	Gestion du son de l'application. 3 sons différents : un quand on pose un cube, un quand on le détruit et un son principal.
Sound.hpp	
Cube.cpp	Gestion de la création d'un cube. Structure, initialisation des buffers, dessin d'un cube et fonctions pour gérer l'attribut couleur.
Cube.hpp	
CubeData.hpp	Contient les données du cube telles que la position de ses vectrices et ses index.

Cursor.cpp	Classe qui hérite du cube. Permet à l'utilisateur de pouvoir bouger un curseur qui a la forme d'un cube.
Cursor.hpp	
File.cpp	Gestion de la lecture / écriture des fichiers : pour lire les fichiers texte des points de contrôles, pour écrire les fichiers texte des sauvegardes et enfin pour lire les fichiers texte de chargement de sauvegarde.
File.hpp	
GameController.cpp	Gestion des différentes actions sur le cube et sur la scène.
GameController.hpp	
Geometry.cpp	Fichier d'origine de glimac.
Geometry.hpp	
gl-exception.cpp	Permet la gestion des erreurs de glew grâce au GLCall.
gl-exception.hpp	
Image.cpp	Fichier d'origine de glimac. Permet de charger une image.
Image.hpp	
FreeFlyCamera.cpp	Gestion d'une caméra de type freefly car elle permet de se déplacer librement dans une scène. Déplacement gauche, droite, haut, bas, avant/arrière et rotation.
FreeFlyCamera.hpp	
Menu.cpp	Création d'un menu sous la forme d'une texture. Gestion des clics sur cette texture.
Menu.hpp	
Overlay.cpp	Implémentation de l'interface graphique que fournit Imgui. Gestion des clics et événements associés aux fonctionnalités.
Overlay.hpp	
Program.cpp	Fichier d'origine de glimac. Objet opengl qui résulte de la compilation des shaders.
Program.hpp	
PlayerController.cpp	Gestions des actions de la caméra et des états du jeu.
PlayerController.hpp	
rbf.cpp	Gestion du calcul d'une radial basis function. Récupère le type de rbf utilisé, calcule omega et applique l'interpolation.
rbf.hpp	
Scene.cpp	Gestion de la construction de la scène et de son affichage. Permet de charger et utiliser un programme Opengl. Gestion des uniformes matrices et de la luminosité.
Scene.hpp	
SDLWindowManager.	Fichier d'origine de glimac modifié. Permet de créer une fenêtre

cpp	SDL, de créer un contexte OpenGL et de gérer les swap de buffer.
SDLWindowManager.hpp	
Shader.cpp	Fichier d'origine de glimac. Gère le chargement des shaders.
Shader.hpp	
Texture.cpp	Gestion du chargement des textures.
Texture.hpp	

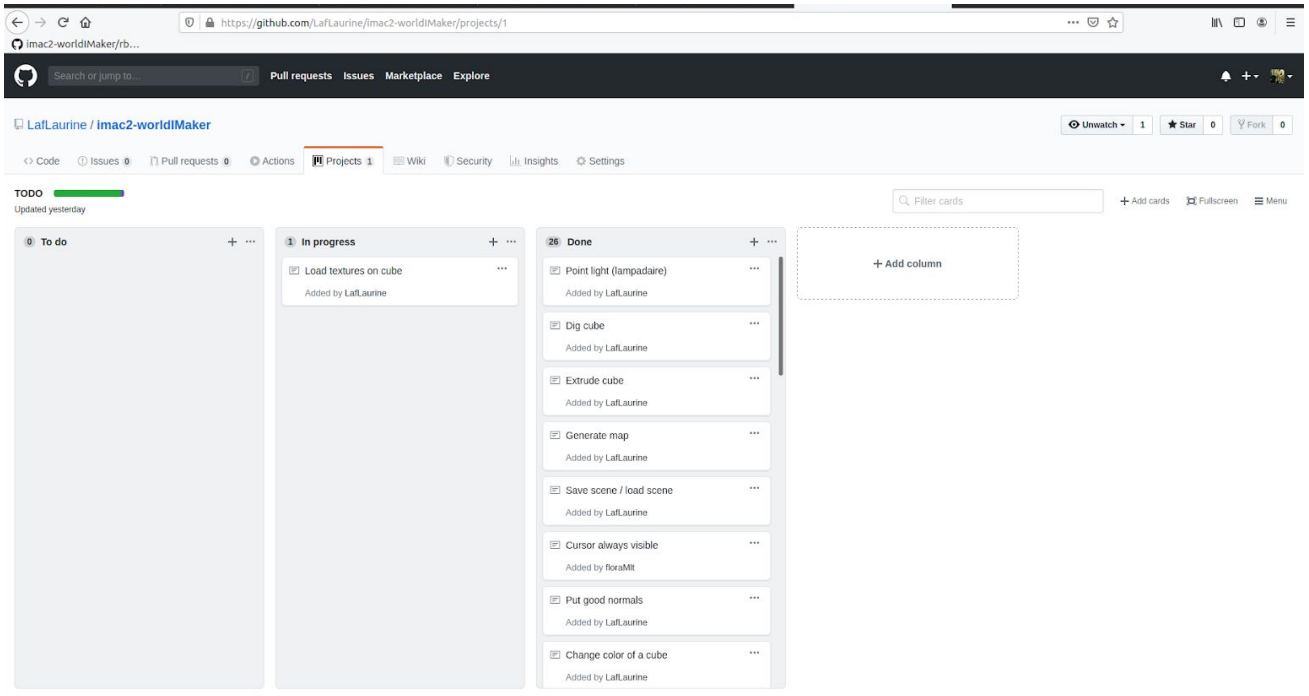
Il y a trois autres fichiers originaux de glimac : stb_image.h, tiny_obj_loader.cpp, tiny_obj_load.h.

ORGANISATION DU PROJET

Tout projet mérite une organisation méticuleuse !

Nous avons d'abord créé un document **Drive** afin de nous organiser et prioriser les tâches à réaliser.

Tout au long du projet, nous nous sommes régulièrement concertées par rapport à son avancée. En particulier, grâce à l'utilisation de **Github** qui nous a grandement aidé pour ce suivi. Grâce à Github, nous avons la possibilité de créer une sorte de Trello, voici le lien vers celui-ci : https://github.com/LafLaurine/imac2-worldIMaker/projects
Cela nous permettait de vraiment suivre la trace de notre avancée.



De plus, nous nous sommes organisées de nombreux moments pour se retrouver (merci la semaine de projet) afin de coder ensemble. Ainsi, nous avons pu réfléchir à deux aux problèmes rencontrés : cela nous a permis d'avoir plusieurs visions sur un même algorithme. Ces deux points de vue nous ont souvent aidé à nous corriger l'une l'autre.
Lors de ce projet, nous avions un cahier des charges à respecter. Voici ci-dessous, le récapitulatif de ses fonctionnalités réalisées (ou non).

Caméra	Oui	Freefly camera
Affichage d'une scène avec des cubes		
Afficher un cube	Oui	
Afficher des cubes	Oui	
Afficher la scène	Oui	
Edition des cubes		
Sélectionner un cube (curseur)	Oui	Le curseur se déplace avec les flèches directionnelles du clavier
Utilisateur peut changer la couleur d'un cube	Oui	En appuyant sur c, l'utilisateur peut changé le cube avec la couleur choisie dans la roue chromatique des couleurs.
Sculpture du terrain		
Ajouter un bloc	Oui	En appuyant sur le bouton "add cube" l'utilisateur peut ajouter un cube
Supprimer un bloc	Oui	En appuyant sur le bouton "destroy cube" l'utilisateur peut détruire un cube
Extrude	Oui	En appuyant sur e, l'utilisateur peut extruder un cube.
Dig	Oui	En appuyant sur g, l'utilisateur peut creuser un cube.
Ajout lumières		
1 lumière ponctuelle (point de lumière)	Oui	Gestion de la luminosité et de l'emplacement du point de lumière
1 lumière d'ambiance (directionnelle)	Oui	Gestion de la luminosité et de l'emplacement de la lumière directionnelle
2 ambiances (jour, nuit)	Oui	Grâce à un bouton on/off sur la lumière directionnelle
Génération procédurale		
Radial Basis Functions : interpolation	Oui	
Générer la scène	Oui	
Fonctionnalités additionnelles		
Sauvegarder la scène	Oui	L'utilisateur peut entrer le chemin du fichier et son nom via l'interface et appuyer sur le bouton "save" pour sauvegarder la scène
Charger la scène	Oui	L'utilisateur peut entrer le chemin du fichier et son nom via l'interface et appuyer sur le bouton "load" pour charger la scène
Blocs texturés	Oui	
Musique	Oui	
Menu	Oui	Menu au début, peut commencer une nouvelle scène où charger la dernière sauvegardé. Menu pause en appuyant sur p
Amélioration changer la couleur d'un cube	En partie	En restant appuyé sur C, l'utilisateur peut déplacer le curseur pour peindre plusieurs cubes.

FONCTIONNALITÉS DE L'APPLICATION

Déroulement de l'application :

Au lancement de l'application, un objet de type "Scène" va être instancié et va contenir tous les éléments liés à la scène comme les cubes, leur affichage, etc.
L'application va également créer une fenêtre "overlay" s'occupant du contexte ImGui. En fonction du type de cube qu'il va falloir afficher, l'application va créer un programme associé. Ainsi, chaque type de cube aura son propre programme. Ce programme sera également associé aux shaders correspondant au type de cube.
L'objet scène permet d'initialiser les cubes, en effet les cubes seront créés à l'intérieur de la scène et dessinés par celle-ci. Une matrice permet de stocker les cubes avec leur position et leur longueur/largeur. Puis, les cubes sont sélectionnés dans les matrices et dessinés un par un.

L'utilisateur peut avec la caméra, avancer, reculer, monter, descendre et zoomer/dézoomer. A chaque fois qu'une de ces actions est faite, la matrice view est recalculée.

Utilisation de l'application

Lorsque vous lancez l'application, vous arrivez sur le menu. Soit vous décidez de commencer une nouvelle scène, soit vous chargez la dernière sauvegarde (qui doit s'appeler world.txt et être dans ./asset/doc).

Caméra	Curseur	Cube	Scène
Zoom avant : z Zoom arrière : s Gauche : q Droite : d Rotation haut : u Rotation bas : w Monter : a Descendre : x	Haut : Flèche directionnelle Bas : Flèche directionnelle Gauche : Flèche directionnelle Droite : Flèche directionnelle Avant : + Arrière : -	Ajout : bouton add cube Suppression : bouton destroy cube Reset : bouton reset Ajouter texture : bouton add texture Supprimer texture : bouton delete texture C : changer couleur cube Maintenir C plus flèche directionnelles possible Sol : bouton "set ground" Creuser un cube : G Extrude un cube : E	Sauvegarde fichier : remplir les champs textes puis bouton "save" Chargement fichier : remplir les champs textes puis bouton "load" Générer RBF : bouton "generate rbf" ou "generate big cube"

Affichage d'une scène avec des cubes

Avant de parler de scène contenant des cubes, il faut parler de comment est géré un seul cube.
Un cube est composé de 24 vectrices (4 par face). Comme nous faisons du rendu indexé, nous avons besoin de 36 index par cube.
Pour stocker les vectrices de notre cube, nous utilisons la structure ShapeVertex que nous avons déjà eu l'occasion d'utiliser en TP. Elle est stockée dans CubeData.hpp. Notre gestion de celles-ci aurait pu être beaucoup plus élégante en ne stockant qu'une seule face et en la transposant pour faire les autres.
Les index sont aussi stockés dans le CubeData.

Le cube peut être construit avec une position en paramètre ou alors avec une position et une couleur. Quand il est construit, son buffer est initié (vao, vbo, ibo, vertexattribpointer, …).

Pour dessiner un cube, cela dépend de son type. S'il est de type 0, il n'est pas texturé et donc on le dessine simplement. S'il est de type 1, alors c'est un cube texturé et

on lui bind la texture avant de le dessiner. Chaque cube est dessiné en mode transparence pour avoir une meilleure visibilité de notre scène.

Maintenant, comment dessiner plusieurs cubes ? C'est là qu'intervient la création de notre scène.

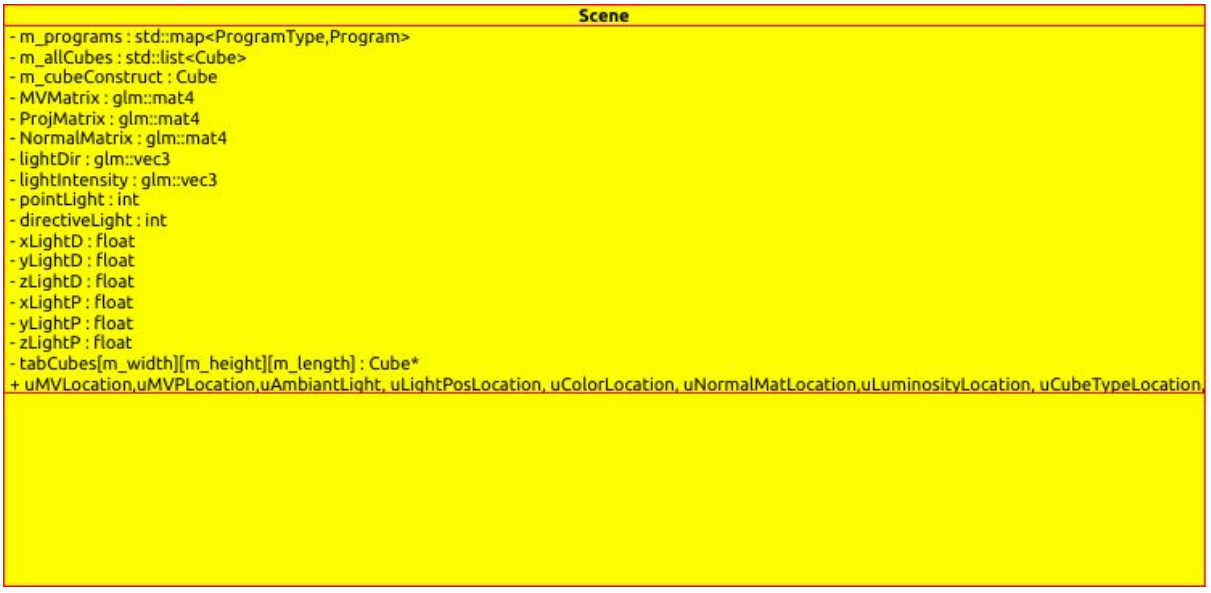


Diagramme de classe de la scène

En commençant le projet, nous nous sommes demandé : "Comment charger plusieurs shaders et gérer plusieurs variables uniformes afin de pouvoir gérer la possibilité d'utiliser ces shaders pour différents éléments ?"
Céline nous a bien aidé sur la question. Grâce à elle, nous avons pu créer une map entre le Program et son type de programme dans la structure de la scène.
Il existe deux types de programmes : Cube ou Menu. Cube va être le programme qui contient les shaders compilés pour les cubes colorés/texturés et Menu est celui pour le menu principal et pause. Grâce à cela, on peut utiliser des shaders sur un objet précis plus facilement car chaque shader n'est pas associé aux mêmes variables uniformes.

Notre monde fait 20 * 20 * 20. Mais la base de la scène est uniquement fait avec l'axe x et z. C'est-à-dire que la scène génère 400 cubes au début pour créer le sol.

La première chose que l'on a fait est de créer notre scène, charger ses shaders associés et l'utiliser. Ensuite, on récupère les uniformes matrices de notre shader en fonction du type de notre programme. On définit ensuite notre matrice modèle, notre matrice de vue et notre matrice de projection, notre matrice normale et notre matrice de projection vue qui est égale à MV * ProjMatrix.

Notre scène est ensuite initialisée. La fonction initAllCubes contenue dans le GameController parcourt la scène et ajoute les cubes à celle-ci.

```
void GameController::initAllCubes() {
    //initialize all the cube of the scene
```

```
        for (int z = 0; z < m_scene->getLength(); z++) {
            for(int x= 0 ; x< m_scene->getWidth() ; x++)
            {
                m_scene->getAllCubes().push_back(glm::ivec3(x,0,z));
            m_scene->tabCubes[x][0][z] = &m_scene->getAllCubes().back();

            }
        }
    }
```

Comme les cubes sont initiés, on peut maintenant les dessiner. On parcourt la liste des cubes et on utilise la fonction recalculateMatrices contenue dans la scène afin de pouvoir calculer la MV en fonction de la position du cube et de la caméra.

```
    void Scene::recalculateMatrices(FreeFlyCamera &camera,Cube cube) {
        //compute the model view matrix with the camera
        float xCube = cube.getPosition().x;
        float yCube = cube.getPosition().y;
        float zCube = cube.getPosition().z;
        glm::vec3 cubePos = {xCube, yCube, zCube};
        glm::mat4 camera_VM = camera.getViewMatrix();
        glm::mat4 modelMat = glm::translate(glm::mat4(1.0f), cubePos);
        glUniform3fv(uColorLocation, 1,
glm::value_ptr(cube.getColor()));
        glUniformMatrix4fv(uMVLocation, // Location
                           1, // Count
                           GL_FALSE, // Transpose
                           glm::value_ptr(modelMat * camera_VM)); //
Value

    }
```

La caméra utilisée est de type FreeFly afin de pouvoir se déplacer librement.

Afin de générer une meilleure expérience pour l'utilisateur, nous avons créé des fonctions setGround et cleanScene afin de pouvoir ajouter ou supprimer le sol de base.

Edition des cubes

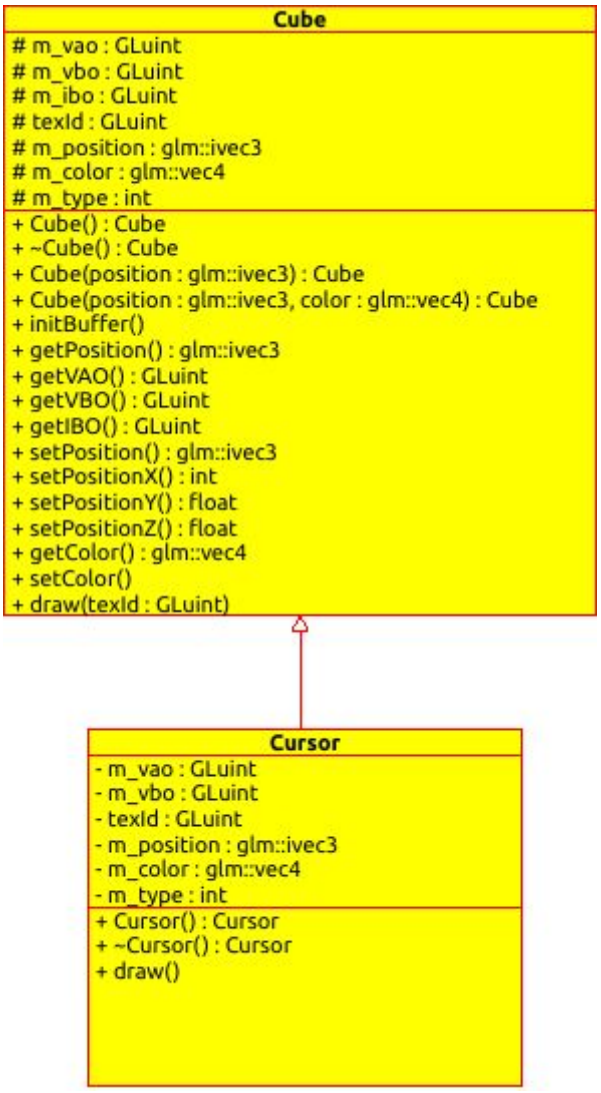


Diagramme des classes cube et cursor, qui montre l'héritage

La sélection du cube s'effectue grâce au curseur. Le cube est sélectionné dès que le curseur est par-dessus celui-ci. La sélection s'effectue dans le GameController. La fonction checkPositionCursor est de type bool et prend en paramètre la position du curseur. Cette fonction permet de vérifier si le curseur est bien dans les limites du monde. La fonction isThereACube est de type bool, et comme son nom l'indique, elle vérifie s'il y a un cube à la position du curseur ou non.

Ensuite, nous avons checkCurrentCube qui regarde quel est le cube qui est à l'endroit où se trouve le curseur. Elle utilise le tableau comportant les pointeurs vers les cubes avec la position du curseur et renvoie un pointeur vers le cube en question, ou s'il n'y a pas de cube un nullptr. Cette fonction été très utilisée pour mettre à jour le CurrentCube, en effet quand on ajoutait un cube à partir du curseur dans un emplacement vide, il fallait mettre à jour le currentCube pour qu'il corresponde au cube juste ajouté et inversement avec la suppression de cube avec le curseur.

Puis, nous avons quelques méthodes liées au curseur et à son déplacement. Tout d'abord addToCursor qui permet d'ajouter un cube à l'endroit où le curseur se trouve. Cette méthode regarde dans currentCube s'il y a un pointeur ou si celui-ci est null. Puis, si le pointeur est null, elle crée un nouveau cube et lui donne les position du curseur, elle passe ensuite le cube à addCube qui se charge d'ajouter le cube à la liste et au tableau. Enfin, elle actualise la sélection avec checkCurrentCube. La méthode deleteToCuror permet quant à elle de supprimer un cube depuis le curseur et utilise les méthode deleteCube et checkCurrentCube.

Enfin, pour le curseur en lui-même, nous avons choisi de mettre en place deux méthodes différentes quant à son déplacement. Tout d'abord une méthode updateCursorPosition qui donne un nouvelle position au curseur, la position est prise en argument et remplace l'ancienne.
Puis, une méthode movecursor qui prend le vecteur 3 reçu en argument et qui l'ajoute à la position actuelle du curseur, ce qui correspond plus à un déplacement de curseur. On rappelle après ces changements de position de curseur la méthode CheckCurrentCube pour actualiser l'attribut currentCube.

Pour changer la couleur d'un cube, nous avons inclu l'Overlay dans le GameController.
La fonction change color cube prend paramètre l'overlay par référence. On récupère un pointeur sur le cube à la position du curseur. S'il y a effectivement un cube, on récupère la valeur du color picker de l'overlay et on la transfère au cube.

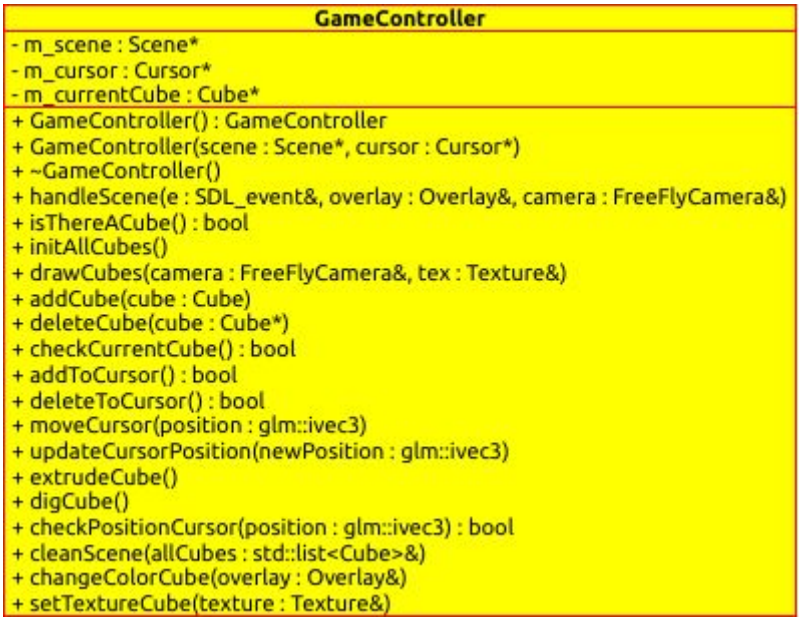


Diagramme de la classe GameController

Sculpture du terrain

La sculpture du terrain se fait à l'aide des boutons présents dans l'interface imGui et des touches du clavier.

Tout d'abord, l'utilisateur peut ajouter un cube à la scène en appuyant sur le bouton Add cube. Ce bouton est relié à la méthode addToCursor qui s'occupe en premier lieu d'appeler la méthode de vérification de cube isThereACube afin d'éviter d'ajouter un cube à un endroit déjà occupé par un autre. Puis, s'il n'y a pas de cube à l'emplacement souhaité, la méthode appelle AddCube et lui passe en argument un nouveau cube ayant comme position celle du curseur. Enfin la méthode addCube ajoute le nouveau cube reçu en paramètre dans la liste et son pointeur dans le tableau.

L'utilisateur peut également supprimer un cube de la scène grâce au bouton Delete cube. Ce bouton est relié à la méthode deleteToCursor qui va tout d'abord appeler isThereACube (méthode de vérification) pour éviter de supprimer un inexistant. Elle va ensuite appeler la méthode deleteCube et lui passe en argument le pointeur vers le cube.

De plus, l'utilisateur peut effectuer un extrude et un dig sur le plus haut cube se situant aux coordonnées x et z du curseur.
La méthode extrude récupère donc la position de curseur et identifie le cube ayant le y le plus grand dans la colonne correspondante en x et z égaux au curseur. Elle parcourt ainsi le tableau du y le plus grand jusqu'au sol si besoin jusqu'à trouver un pointeur qui ne soit pas null, autrement dit jusqu'à trouver un pointeur sur un cube. Puis, une fois le cube trouvé, la méthode updateCursorPosition déplace le curseur à l'emplacement juste au dessus du cube trouvé et appelle la méthode de création de cube addToCursor.
La méthode dig quant à elle parcoure la colonne de la même façon que l'extrude et dès qu'un cube est trouvé, elle appelle la méthode updateCursorPosition qui va placer le curseur à l'endroit du cube trouvé. Une fois le curseur à la bonne position, la méthode appelle deleteToCursor.

Grâce à tous ses outils, Toto a de quoi s'amuser et créer de super scènes !

Génération procédurale

Dans cette application, on peut utiliser cinq types de RBF : Gaussian, ThinPlateSpline, InverseQuadratic, BiharmonicSpline et Multiquadric.
Nous avons choisi de définir nos points de contrôle dans des fichiers .txt, que l'utilisateur peut lui-même créer par la suite.
Chaque cube de la scène va être calculé à partir des points de contrôle.
On utilise la fonction d'interpolation vue en cours, appelée g(x) et qui se note :

$$g(x)=\sum i=1k\omega i*\Phi(|x-xi|)$$

où :
ωi sont les coefficients à déterminer
ϕ(|a−b|) une fonction “radiale” dont la valeur dépend de la distance entre les points a et b.

Le fichier des points de contrôle contient ce qu’il suit :
RBF
position en x position en y position en z coefficient
….

Nous avons décidé de séparer le calcul (dans un fichier rbf.cpp, associé à son rbf.hpp) et le chargement du fichier des points de contrôle (dans file.cpp associé à file.hpp).

Le fichier de points de contrôle est lu dans la fonction readFileControl et permet d’ajouter au vecteur de point de contrôles, les points récupérés par la lecture du fichier. On récupère la position en x, en y, en z et sa valeur (son poids).

Après avoir effectué quelques calculs papiers et avoir parlé avec des camarades, nous avons pu commencer à concevoir notre résolution de rbf.
Soit A une matrice.
On cherche à résoudre un système de type A*omega = poids. Le seul élément non connu étant omega.

Pour trouver omega :

- Créer une matrice (on la remplit de zéro en lui donnant la taille de nos points de contrôles)
- Créer un vecteur de poids qui ne contient que des 1, de la taille de nos points de contrôles
- Associer ces poids aux points de contrôles
- Remplir notre matrice en fonction de la taille du vecteur des points de contrôle
- Décomposer la matrice avec la méthode LU
- Résoudre l’équation A * omega = poids

Une fois que l’on a omega, on va pouvoir appliquer la rbf en fonction de son type. On parcourt notre scène puis notre vecteur de points de contrôle. Pour chaque point, on calcule sa valeur interpolée en fonction de la rbf choisie, enfin on multiplie par omega. On utilise aussi le paramètre epsilon, présent dans chaque type de rbf excepté la ThinPlateSpline. Voir la section maths pour notre choix d’epsilon.

Nous obtenons donc les coordonnées après interpolation. Si la valeur d’interpolation est négative, alors le cube n’est pas dessiné.

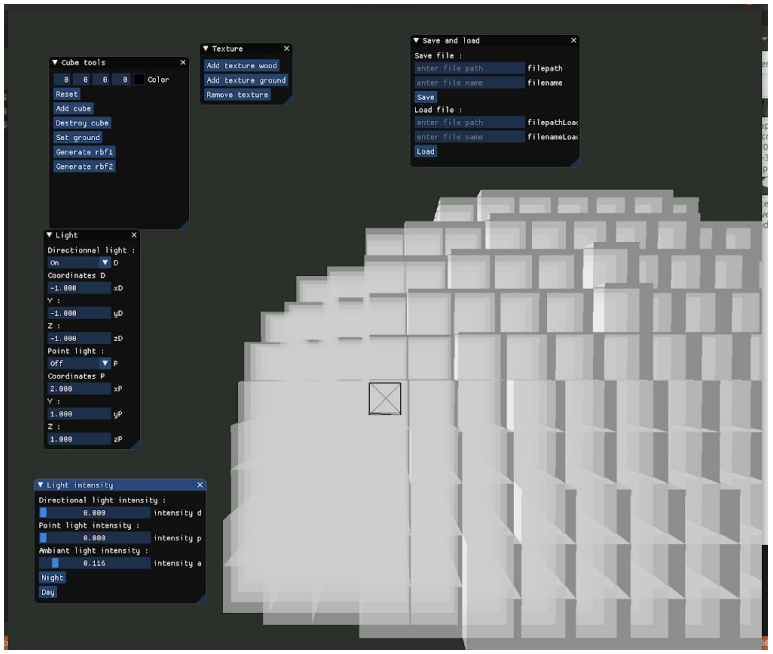
L’algorithme est inspiré de ce repo github et des cours de M.Nozick :
https://github.com/yuki-koyama/rbf-interpolator/blob/master/rbf_interpolator/interpolator.cpp

Ajout de lumières

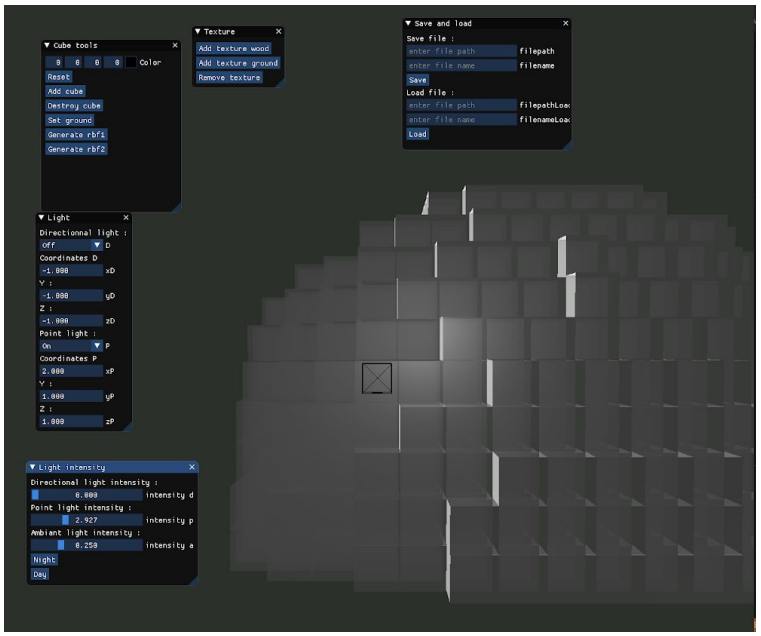
Les lumières sont gérées dans le shader colorCube.fs.glsl. Pour la lumière directionnelle et le point, on utilise l'éclairage blinnPhong. C'est une bonne approximation de la lumière. Pour cela, on a dû ajouter des uniformes à notre shader. La fonction addLight contenue dans la scène permet de récupérer ces variables uniformes et de leur envoyer des valeurs.
Le mode de jour/nuit peut se faire grâce à la gestion de l'intensité des lumières. On peut aussi activer le bouton jour ou nuit.
Nous avons aussi fait en sorte que l'utilisateur puisse déplacer les deux lumières à sa guise sur les axes x,y et z.
En plus de ces deux lumières, nous avons aussi une lumière d'ambiance afin que même si les deux précédentes sont désactivées, on puisse avoir une petite source de lumière. On peut aussi modifier l'intensité de celle-ci.
Ces fonctions de modification sur la lumière sont contenues dans la scène.

Le total de la lumière calculée est donc le minimum entre les lumières de blinnPhong et 1.

```
vec3 totalLuminosity = min(lum + ambiantLightIntensity, 1.);
```



Mode lumière directionnelle et point activé



Mode lumière directionnelle désactivé et point activé

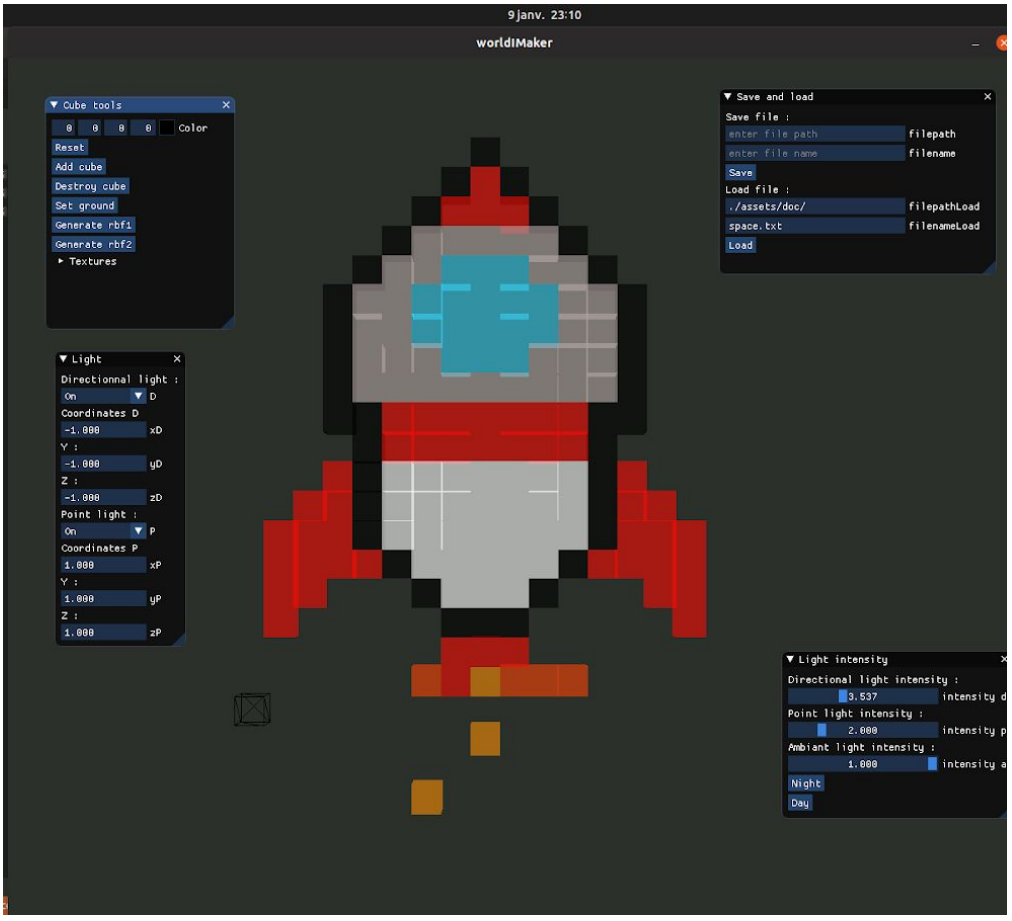
Sauvegarde / Chargement de la scène

Pour gérer la sauvegarde et le chargement, on utilise des fichiers .txt. Ces actions sont gérées dans le File. On retrouve donc dans celui-ci la fonction saveFile et la fonction loadFile. Elles prennent en paramètre le chemin vers le fichier, le nom du fichier et la liste des cubes présents dans la scène.
La stratégie est la même pour les deux fonctions, seulement on écrit le fichier pour sauvegarder la scène et on lit le fichier pour charger la scène.
L'existence des fichiers est vérifié.

Pour la sauvegarde, on vérifie si le fichier est bien créé et on va parcourir la scène grâce à un for each lambda, on capture le fichier et on récupère la position en x,y et z de chaque cube ainsi que sa couleur associée. On ferme ensuite le fichier et on affiche à l'utilisateur que son fichier a bien été sauvegardé.

Pour le chargement, on commence par nettoyer la liste de cube (cf : std::<list> Cube allCubes) qui contient la liste des cubes de notre monde actuel. Ensuite on ouvre le fichier et on récupère les éléments à chaque ligne. On ajoute la position et la couleur au cube et on ajoute celui-ci à notre liste de cubes. On l'ajoute ensuite à notre tableau de cube contenu dans la scène.

```
scene.tabCubes[cube.getPosition().x][cube.getPosition().y][cube.getPosition().z] = &allCubes.back();
```



Blocs texturés

La texture est directement chargée dès l'appel au constructeur. Un cube comporte dans ses attributs un GLuint de l'id de la texture.

Malheureusement, nous n'avons pas réussi à sauvegarder et charger la texture. La fonction setTextureCube du GameController prend en paramètre une texture par référence. Cette fonction récupère le cube sur lequel on est, puis, si le type du cube est à 0, alors on peut lui appliquer une texture. Le type du cube devient donc 1 et on lui passe l'id de la texture voulue.

```
cubePtr->setCubeTextureId(tex.getId());
```

Pour enlever une texture, on change juste le type du cube à 0 (via la fonction removeTextureCube).

Du coup, que se passe-t-il dans le dessin lorsque le type est à 1 ? La texture choisie est bindé au cube !

Nous avons réfléchi entre faire des shaders séparés ou non pour appliquer la texture, mais finalement nous avons adopté la solution suivante : nous passons dans le fragment shader une variable uniforme de type bool pour savoir quand la texture est

appelée ou non. On passe aussi une variable uniforme pour connaître le type du cube (donc s'il est texturé ou non). On effectue l'opération suivante :

```
if(setTexture && uCubeType == 1) {
        fragColor = vec4(tex.rgb * totalLuminosity, 1.0f);
    }
    else if((setTexture && uCubeType == 0) || (!setTexture && uCubeType
== 0)){
        fragColor = vec4(totalLuminosity  * color, 1.0f);
        fragColor.a = 0.6;
    }
```

On voit donc que l'on applique la texture uniquement si le type nous le permet.
En dessinant les cubes, on n'oublie pas de gérer les glUniformli associés à ces variables uniformes.

Mais, comment choisissons-nous la texture voulue ?
On choisit la texture via l'interface ImGui. Les textures étant toutes chargées au début de l'application, celle choisie via le bouton va être appliquée.

Musique

Alors oui, on aurait pu se passer de cette fonctionnalité pour alléger notre application, mais que serait notre application sans de petits sons de Minecraft ? Toto va pouvoir être fier de montrer ses créations (sous risque d'exposition aux droits d'auteur, mais grâce à sa superbe application, il va faire de beaux modèles et devenir riche).

La musique est gérée grâce à la librairie SDL2_MIXER. Pour bien la faire fonctionner, on a dû modifier le CMake. On utilise deux autres CMAKE : FindSDL2_mixer.cmake et FindSDL2.cmake. Sans ces deux fichiers externes, nous n'arrivions pas à faire fonctionner SDL_Mixer.

Nous utilisons trois sons : la musique de fond, le son de l'ajout de cube et le son de destruction de cube. Ces types de son sont stockés dans un enum.
La musique de fond est jouée dès le début du programme. Quand on ajoute un cube, on joue le son de type BUILD. Quand on détruit un cube, c'est le son de type DESTROY qui est joué.

Menu

Quid du menu ? Pourquoi avoir fait cela ? La réponse est simple : compliquer un projet qui est déjà assez conséquent. Effectivement, la mise en place d'un menu à été légèrement long dû à des petites erreurs sur la texture.
La gestion de celui-ci n'est pas non plus très bien faite. Le système de menu permet de gérer l'état du jeu, donc celui du GameController. Le jeu peut : commencer directement en

chargeant la scène par défaut, charger une scène (world.txt), se mettre en pause (en appuyant sur P).

On définit des vertices pour faire un rectangle de la taille de notre fenêtre. Cette fois-ci, on utilise la structure VertexTex qui comporte un vec2 pour la position et un vec2 pour la texture.
On crée aussi les indices associés pour dessiner de façon instancié.

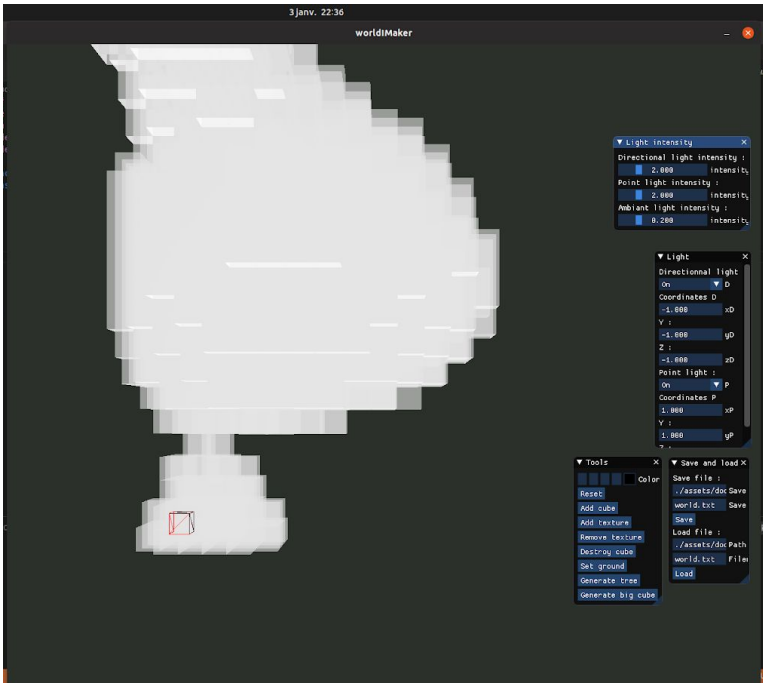
Le menu utilise des shaders autres que ceux du cube qui ne contiennent qu'un uniform sampler2D pour la texture ainsi qu'un fFragColor. Lorsqu'une instance du menu est créée, on lui envoie en paramètre la scène, le type du program et le nom de la texture. Le constructeur va créer le vbo, vao et ibo associés à ses positions et coordonnés de la texture. On initialise donc son buffer.

La fonction floatIsBetween(value,min,max) vérifie si la valeur est bien contenue dans l'intervalle [min,max]. Cela nous permet de gérer les clics sur le menu. Comme le GameController possède plusieurs états, le menu n'est pas tout le temps dessiné.

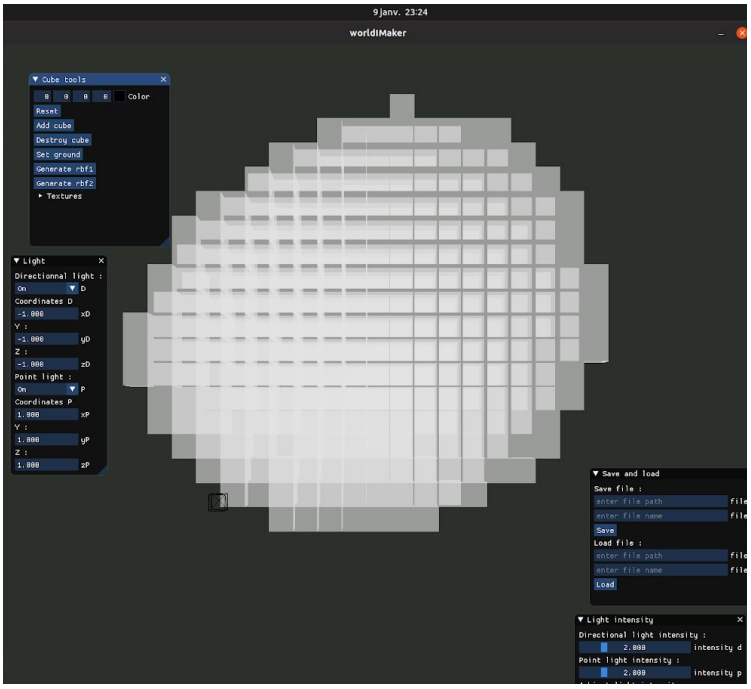
On commence avec l'état gameOn à 0 et on dessine le menu principal. Si l'on clique sur "Create new scene", alors gameOn passe à 1 et on commence normalement.
Si l'on clique sur "Load last scene" alors l'état gameLoad passe à 1 et le jeu charge la sauvegarde. Ensuite, l'était gameOn passe à 1 pour démarrer le jeu.
Maintenant, si l'on appuie sur P pendant l'état gameOn, le jeu passe à l'état gamePause et le jeu est en pause. Si l'on appuie de nouveau sur P alors l'état passe à gameOn.

(on aurait pu vraiment implémenter tout cela avec le design pattern state en créant une interface, mais les états sont trop peu utilisés pour cela).

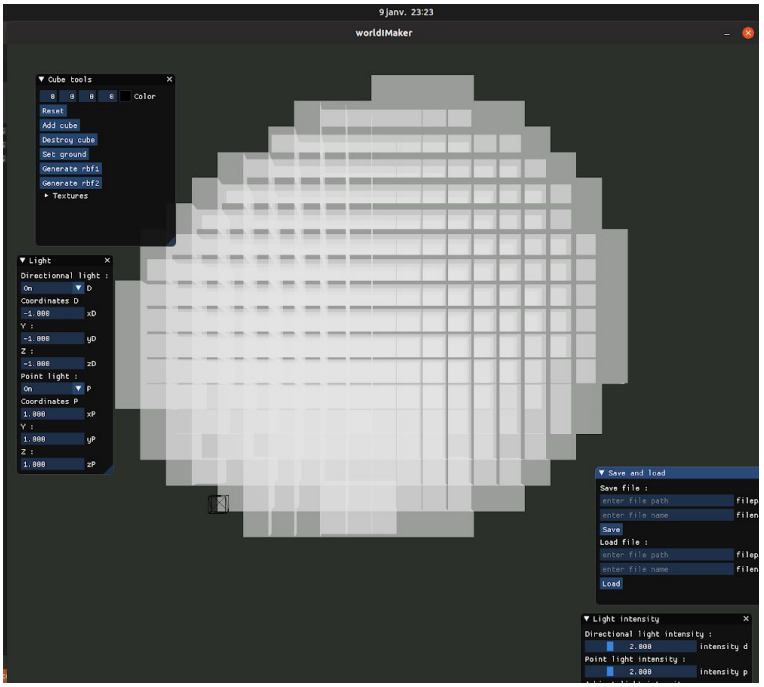
Maths : résultats obtenus par génération procédurale



Arbre généré avec la radial basis function InverseQuadratic et le fichier “controls.txt”. Reste en paix mon arbre, un ami parti trop tôt. En effet, cet arbre a été généré avant la déforestation (avant que nous changions notre algorithme de cube).

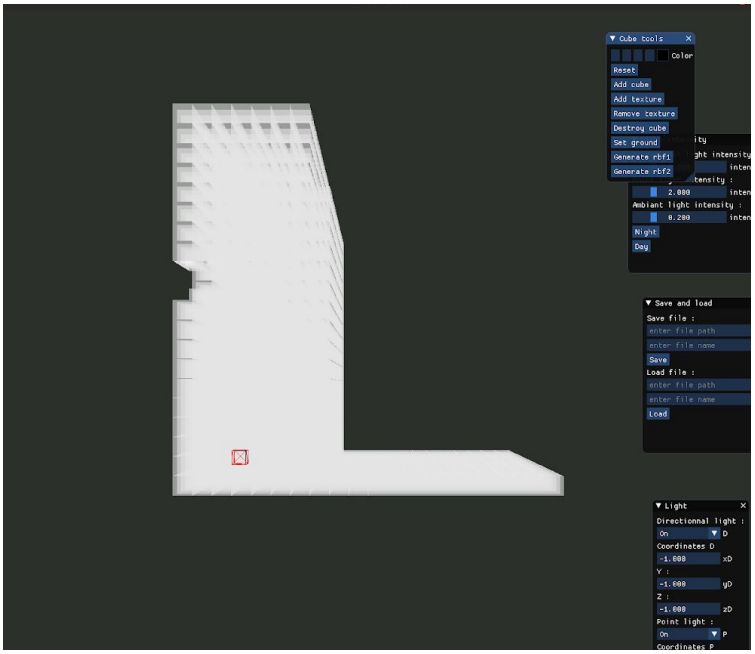


RBF multiquadric, epsilon = 0.4

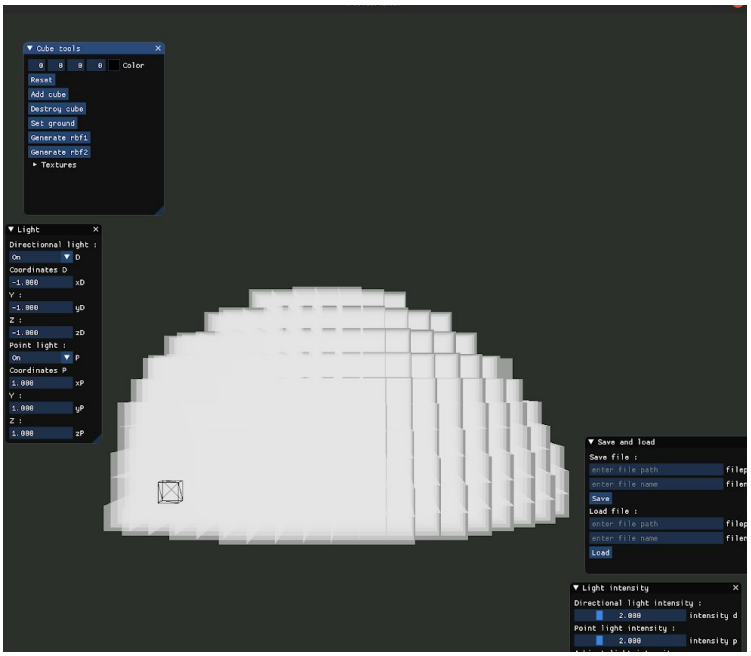


RBF multiquadric, epsilon = 1

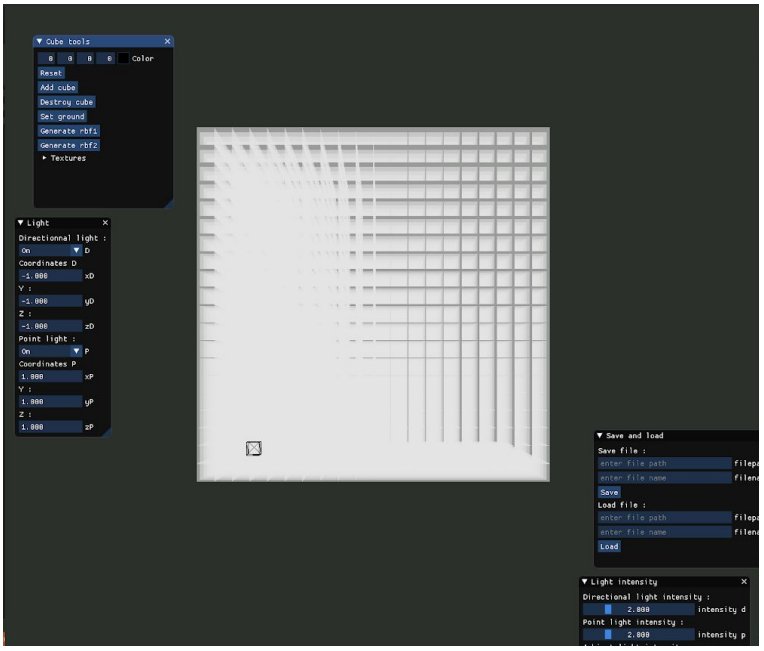
En observant plusieurs fois la même fonction avec les mêmes points de contrôles pour des epsilon différents, on en a déduit que epsilon gère le relief de la scène. Nous avons donc choisie de garder epsilon = 0.8 afin d'avoir un relief ni trop présent ni trop absent.



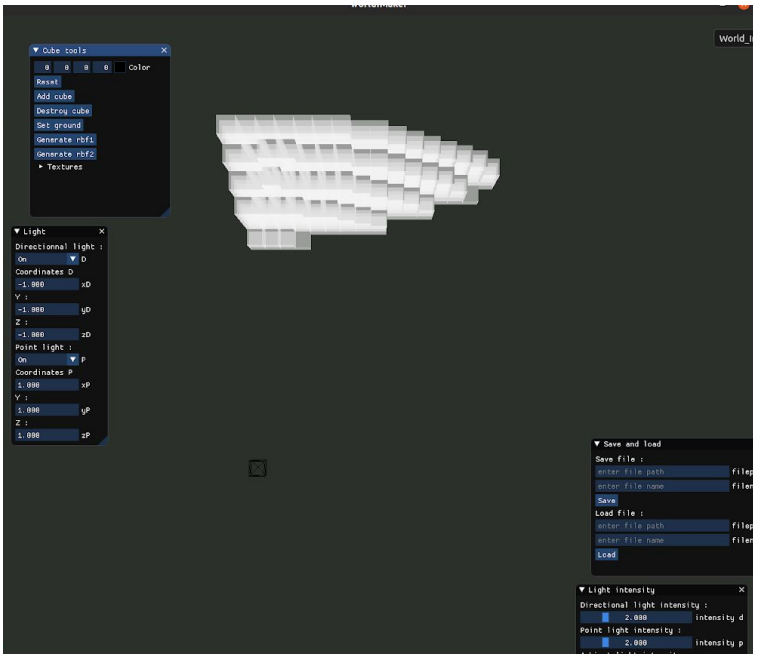
RBF Gaussian



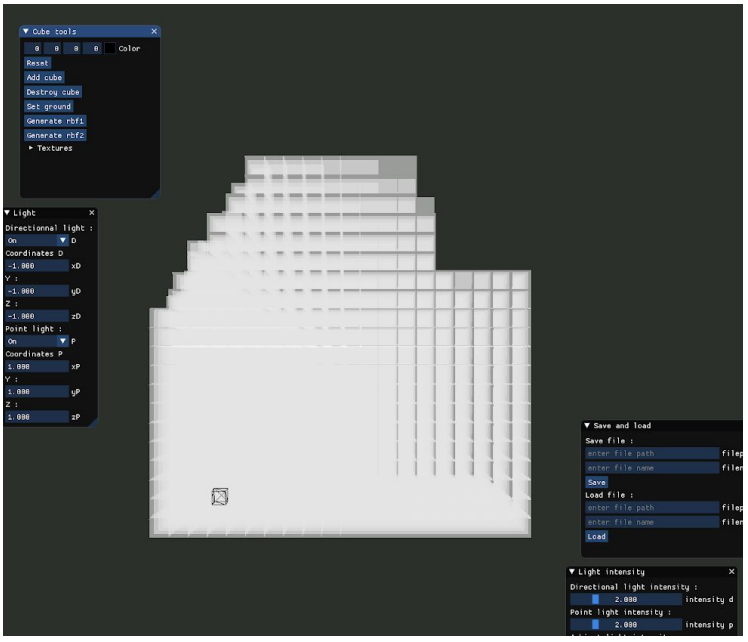
RBF multiquadric



RBF Biharmonic Spline



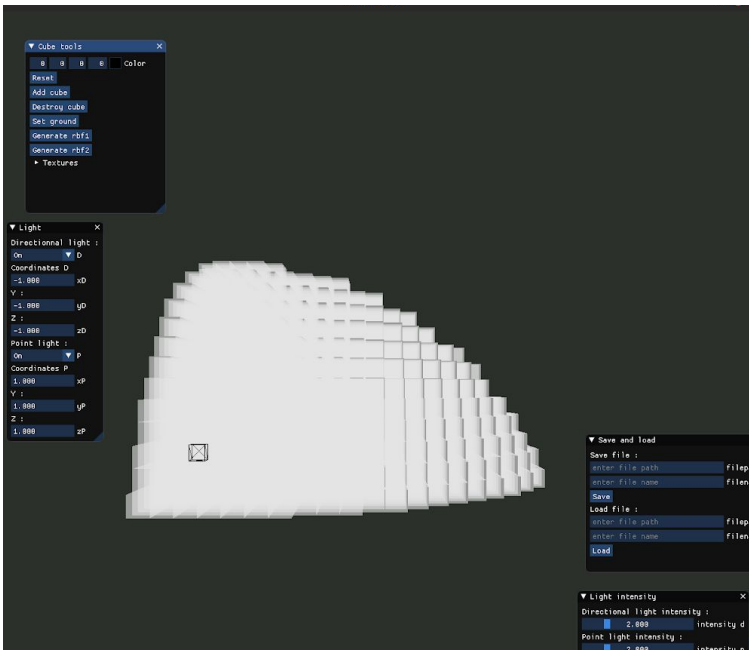
RBF ThinSplateSpline



RBF Gaussian

On s’est rendues compte que pour faire des scènes de plutôt typé “montagne”, “îles”, etc, il fallait plutôt passer par la rbf Inverse Quadratic ou Gaussian.

Ce se donc vers ces choix que nous nous sommes tournés, nous avons préféré utiliser les radiales décroissantes inverse quadratic et gaussian.



RBF Gaussian

DIFFICULTÉS RENCONTRÉES

Par rapport à la texture, nous avons pendant un moment eu des problèmes avec nos lumières qui faisaient une surexposition sur la texture, ce qu'il fait que l'on ne voyait pas si l'on avait vraiment réussi à l'appliquer ou non. Sans faire attention, nous multiplions mal notre matrice ce qui a causé le fait d'avoir une couleur unie à la place de notre texture. On a résolu le problème en corrigeant nos shaders et nos calculs matriciels.

Passer du vector à la list a été assez compliqué à cause des changements que cela provoquait après dans les diverses méthodes l'utilisant. En effet, quand le changement a été effectué, les fonctionnalités comme ajouter un cube ou encore sauvegarder et charger une scène ne fonctionnaient plus et le travail de debug a été assez long, mais très enrichissant. Le changement de conteneur, mais aussi la suppression de la logique visible/invisible nous a poussé à changer entre autre les algorithmes de parcours et les actions réalisées sur les conteneurs. Enfin, les radial basis function on été compliquées au début à debugger, mais à force de réflexion et de persuasion, nous avons réussi à tout réimplémenter (rip le petit arbre quand même).

Interférences entre sdl event et imgui event :

Malheureusement, à chaque événement SDL ou ImGui, les deux peuvent s'entremêler et causer quelques problèmes. Par exemple, on a vécu le fait que le loadFile s'est déclenché tout seul pendant que l'on déplaçait la caméra (même si on utilise ImGui_Prevent pour empêcher les événements de se déclencher)

HISTORIQUE ET CHANGEMENTS

Au début du projet, nous avions choisi de stocker les cubes dans un vecteur afin de pouvoir accéder facilement à tous nos cubes et rendre le tout plus dynamique.
De plus, tous les cubes de la scène étaient initialisés au début du programme et un booléen lié à la visibilité des cubes indiquait si le programme les dessinait ou non. Ainsi, il ne nous restait plus qu'à mettre le cube en visible ou invisible pour le voir ou non dans la scène.
Cependant, ce type de fonctionnement commençait à devenir assez gourmand en mémoire dès que la scène était plus grande. De plus, l'extrude et le dig nous posaient quelques soucis, notamment lors de la récupération de l'indice du cube le plus haut d'une colonne.

Nous avons donc décidé de changer la notion de visibilité en mettant tout d'abord dans une liste doublement chaînée seulement les cubes à dessiner. Ces cubes n'auraient ainsi pas à changer d'adresse durant l'exécution, ce qui permettra d'éviter certains problèmes liés à la recherche d'un cube et à son identification. En effet, la liste doublement chaînée ne "déplace pas" les éléments une fois que sa taille a augmenté, à contrario du vecteur.

Cependant, la liste nécessite d'être mise à jour dès qu'un cube est ajouté ou enlevé et cela nous a permis de gagner en utilisation de la mémoire.
De plus, nous avons décidé de mettre en place un tableau multidimensionnel (ici à 3 dimensions, correspondant aux coordonnées x, y et z) afin de pouvoir associer chaque case du tableau correspondant à une position dans la scène, à un pointeur vers le cube stocké dans la liste ayant les mêmes positions. Ce tableau nous a permis entre autre de réussir à trouver le cube dessiné le plus haut d'une colonne afin de pouvoir faire l'extrude et le dig.

Avec cette nouvelle logique, nous avons dû faire attention à la mise à jour du tableau, notamment quand un cube était ajouté ou enlevé de la liste. Sans cette mise à jour, les fonctions comme la vérification d'un cube n'auraient pas bien fonctionné.

CONCLUSION

Après toutes ces belles paroles, nous vous présentons donc WorldIMaker ! Le logiciel d'édition de scène spécialement développé pour Toto. Ce projet a été très enrichissant, il nous a énormément appris, que ce soit en C++, OpenGL, mathématiques ou architecture logicielle. Il y aurait encore beaucoup d'améliorations à apporter !

PARTIE INDIVIDUELLE

LAURINE	FLORA
Réaliser un projet complet à partir de nos connaissances acquises en TP est un réel défi. Mais nous ne sommes pas du tout parti à l'aveugle. Les TPs d'OpenGL sont fait en sorte que l'on puisse acquérir les connaissances nécessaires pour réaliser ce projet. J'ai aimé relever ce défi. Ces derniers temps, je n'avais plus trop confiance en ma façon de concevoir un programme. Grâce à ce projet, j'ai pu reprendre confiance et me rendre compte de mes compétences. Le sujet était très clair et nous permettait de savoir quoi faire et dans quel ordre. Lorsque M.Nozick a dit qu'il allait associer les maths avec le projet d'OpenGL, j'ai eu très peur, mais finalement, j'ai vraiment adoré cette partie. Les RBF c'est très amusant. Avoir réussi à réaliser une application de ce type m'a rendu fière. Je n'arrivais plus à me poser de limites au projet et à m'arrêter. J'ai passé une très bonne semaine de projet avec Flora, où nous avions très bien avancé. Réfléchir à deux sur des algorithmes est une bonne méthode. Enfin, malgré les difficultés nous avons pu avancer en trouvant des solutions. Remerciement à Pierre Thiel et Céline pour leur aide apportée.	Grâce à ce projet, j'ai pu mettre en pratique ce qui nous a été enseigné durant ce semestre, que ce soit en programmation C++, en synthèse d'image ou encore en mathématiques. J'ai pu expérimenter la programmation orientée objet que j'ai beaucoup appréciée. Durant le semestre les TPs de POO et OpenGL m'avaient rendue très curieuse et m'avaient donné envie d'appliquer les notions vues à un véritable projet, ce qui a donc été possible grâce à Toto ! De plus, j'ai beaucoup aimé réfléchir sur la façon dont implémenter les fonctionnalités et comment manipuler les objets et méthodes entre eux. Malgré quelques difficultés au début, et un petit nombre de segmentation fault, j'ai beaucoup apprécié travailler sur ce projet et implémenter de nouvelles fonctionnalités. Enfin Laurine a été une super binôme, dès qu'une interrogation me venait par rapport au projet, je pouvais en parler avec elle et ensemble nous trouvions la façon d'implémenter telle ou telle fonctionnalité au mieux. Ce projet m'a beaucoup appris et je suis très fière de notre travail !