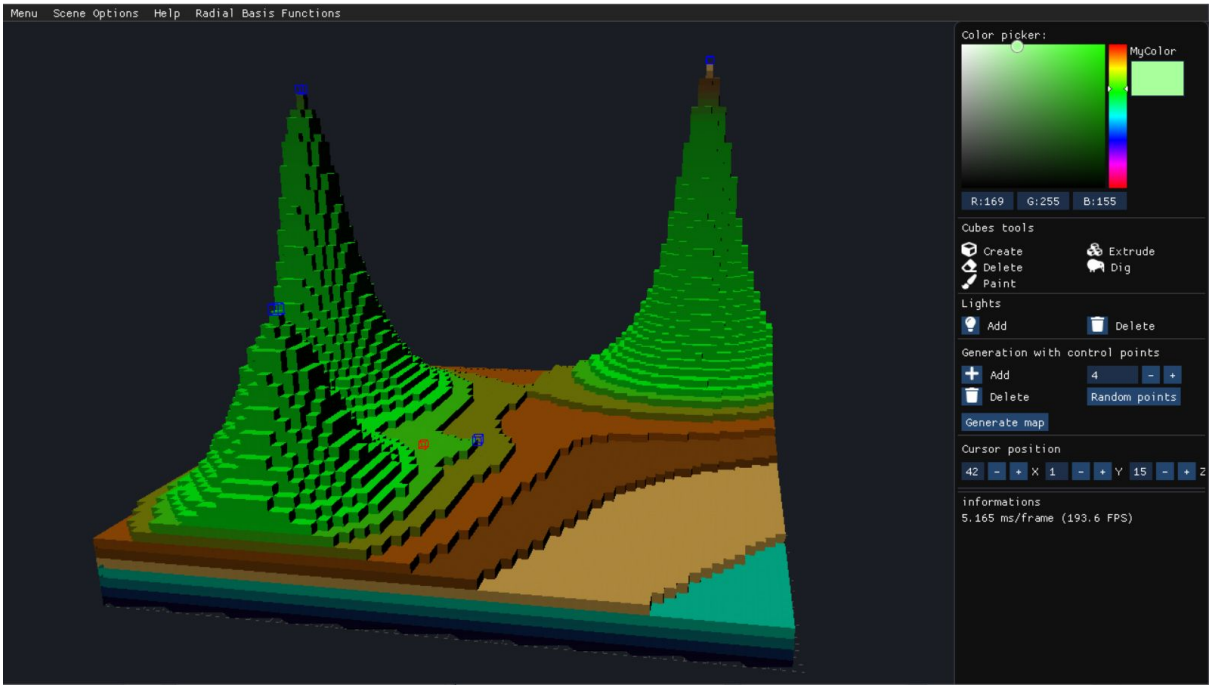
RAPPORT PROJET OPENGL
“SMOOTHIMAKER”

Synthèse d’image
Programmation objet
Mathématiques



Enguerrand De Semet
Amandine Kohlmuller

09.01.2020

Université Gustave Eiffel
Esipe

SOMMAIRE

INTRODUCTION

Dans le cadre des cours de Synthèse d'image, de programmation objet et de Mathématiques, nous avons réalisé un éditeur-visualiseur de terrains et de scènes en 3D. Nous sommes deux, Enguerrand De Semet et Amandine Kohlmuller, et nous avons fait des choix afin d'implémenter un certain nombre de fonctionnalités permettant à l'utilisateur de créer un monde à partir de blocs cubiques et de naviguer dedans.

1. VUE D'ENSEMBLE

1.1 Tableau récapitulatif des fonctionnalités

FONCTIONNALITÉS	FAIT	FONCTIONNE
Représentation en cube (de couleurs et vides)	OUI	OUI
Choix d'une structure de données adapté	OUI	OUI
État initial de 3 cubes de la même couleur à partir du bas	OUI	OUI
Se déplacer dans la scène / Caméra (freefly)	OUI	OUI
Curseur au centre du pavé	OUI	OUI
Déplacer le curseur au clavier	OUI	OUI
Visualisation du curseur (même s'il y a des cubes devant)	OUI	OUI
Interface utilisateur (ImGui)	OUI	OUI
Ajout, édition ou suppression de cubes ("create" & "delete")	OUI	OUI
Outils d'extrusion ("extrude")	OUI	OUI
Outils permettant de creuser ("dig")	OUI	OUI
Génération procédurale à l'aide de radial basis function	OUI	OUI
Gestion d'au moins une lumière directionnelle	OUI	OUI
Gestion d'au moins un point de lumière	OUI	OUI
Mode nuit / mode jour	OUI	OUI

FONCTIONNALITÉS ADDITIONNELLES	FAIT	FONCTIONNE
Amélioration de la sélection	50%	NON
Save and Load	OUI	OUI
Animation de la position du soleil	OUI	OUI
Paramétrage des fonction radiales	OUI	OUI
Interface pour visualiser et gérer la position du curseur	OUI	OUI
Élimination des faces cachées (back face culling)	OUI	OUI
Geometry shaders	OUI	OUI
Affichage d'une scène de 128x128x128, soit 2 000 000 de cubes avec un affichage fluide (au moins 25 fps)	OUI	OUI

1.2 Analyse et Recherches

Dans un premier temps, nous avons réalisé une phase de recherches d'informations sur les différents logiciels existants et les solutions possibles.

En effet, de nombreuses applications comme celle-ci existent déjà. Nous retrouvons par exemple les très connues MagicaVoxel, Qubicle, Goxel, Voxel Max ou encore de plus petits projets comme SimpleVoxelEngine ou polyVox.

Il était intéressant d'avoir une vision globale de ce qui se faisait avant de commencer le projet, ne serait-ce que par curiosité ou afin de ne pas s'éparpiller. Cette étape permet d'avoir une idée générale des choix techniques à mettre en place pour nos contraintes et fonctionnalités.

1.3 Arborescence du programme

En ce qui concerne l'arborescence du projet nous avons fait le choix de diviser le projet en 3 dossiers principaux. Un dossier **/assets** contenant les différentes ressources du projet (shaders , fonts). Un dossier **/libs** contenant les différentes librairies externes au projet. Le dossier principal **/src** contenant tous les fichiers sources de notre projet, ainsi qu'un dossier **/bin** permettant de compiler le projet.

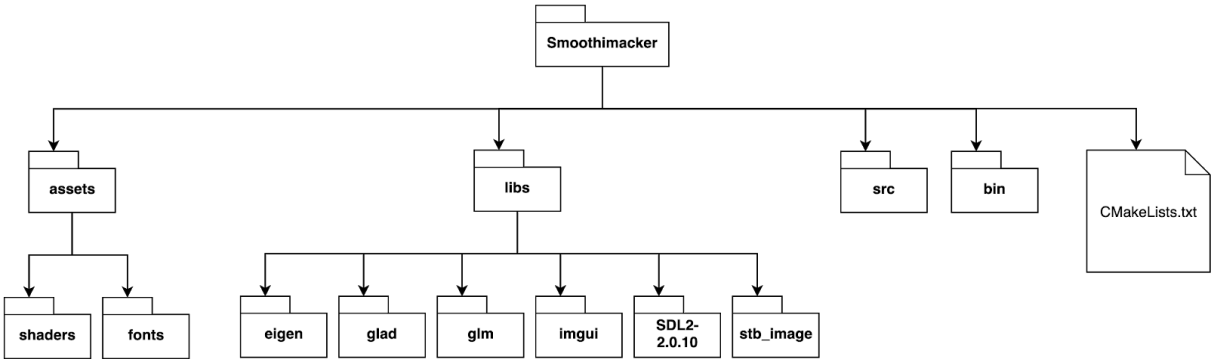


Illustration 1 : Diagramme de l'arborescence générale du projet

Nos fichiers principaux permettant de lancer le projet se trouvent donc à la racine du dossier **src** et nous avons ensuite divisé différentes parties en sous dossiers représentant différents namespace dans notre code.

Les fichiers d'en-tête .hpp associés aux .cpp se trouvent au même emplacement. Ces derniers ne sont pas représentés ici par soucis de clarté.

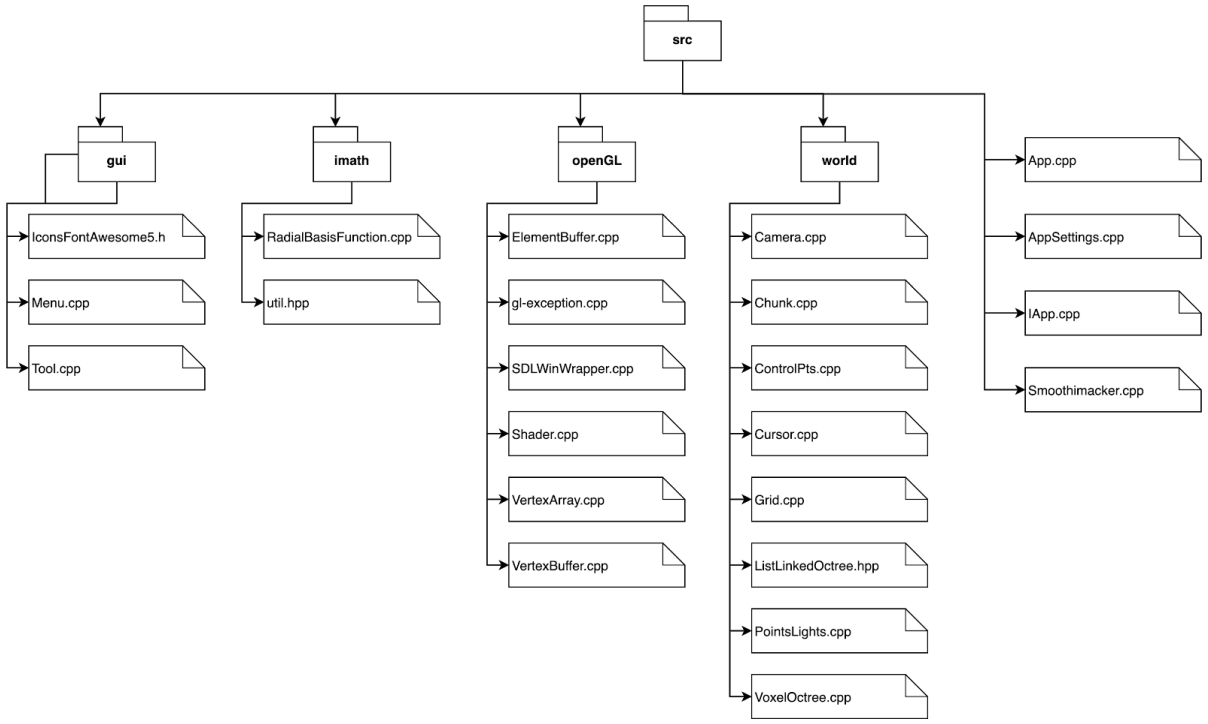


Illustration 2 : Diagramme de l'arborescence du répertoire /src du projet

1.4 Diagramme de classes

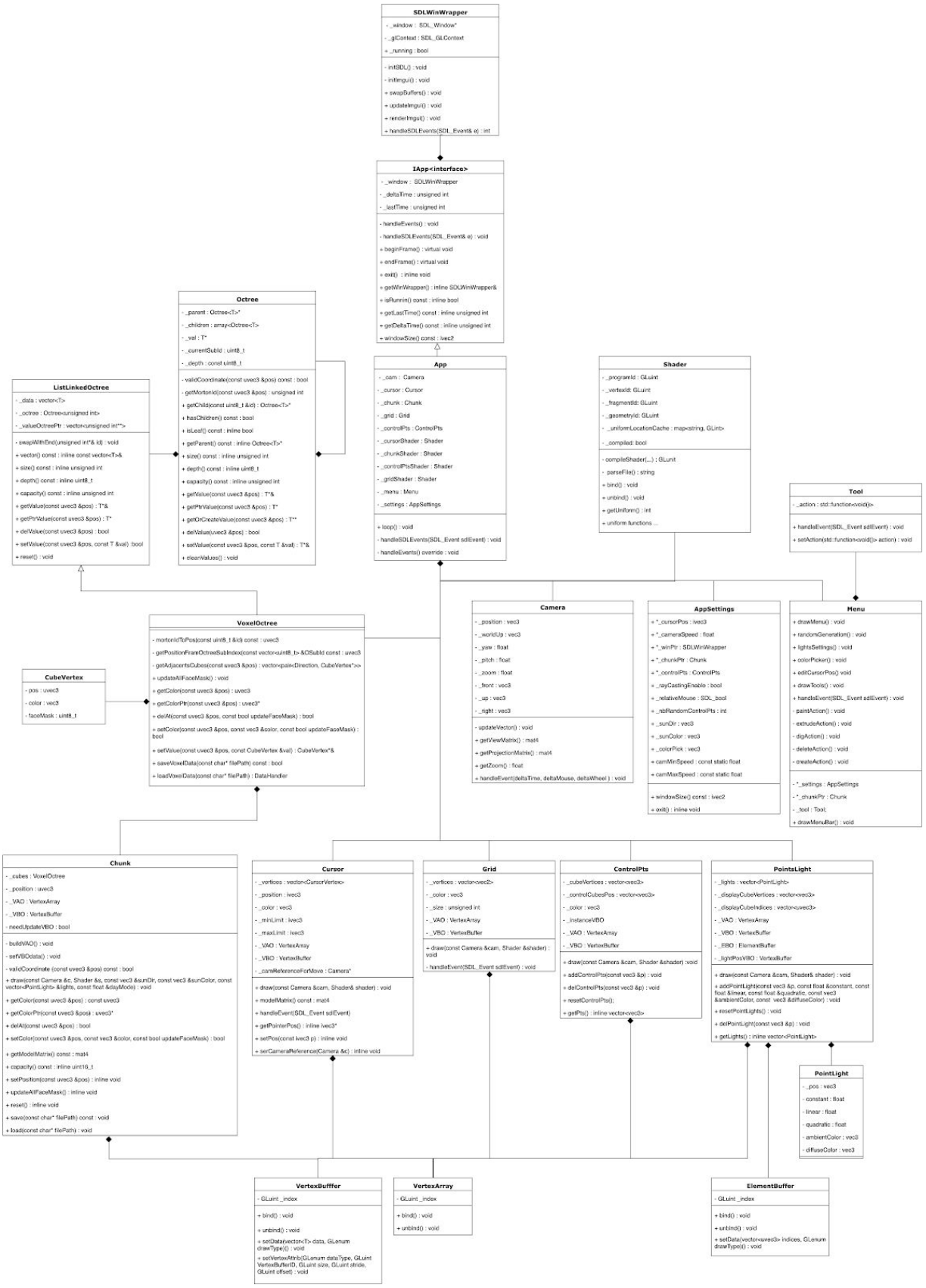


Illustration 3 : Diagramme de classes de notre programme

2. DÉTAIL DES FONCTIONNALITÉS

Nous allons dans les parties suivantes expliquer nos choix d'implémentation en fonction des différentes fonctionnalités de notre projet.

2.1 Abstraction d'openGL

Afin de simplifier l'utilisation d'openGL, nous avons implémenté différentes classes. Ces dernières servent alors d'interface pour abstraire certaines fonctions d'openGl.

La gestion de la compilation des shaders est par exemple caché pour l'utilisateur qui utilise simplement le constructeur avec le chemin du fichier en paramètre.

L'envoi des données à notre carte graphique se fait via la méthode template **setData()** de l'objet VertexBuffer.

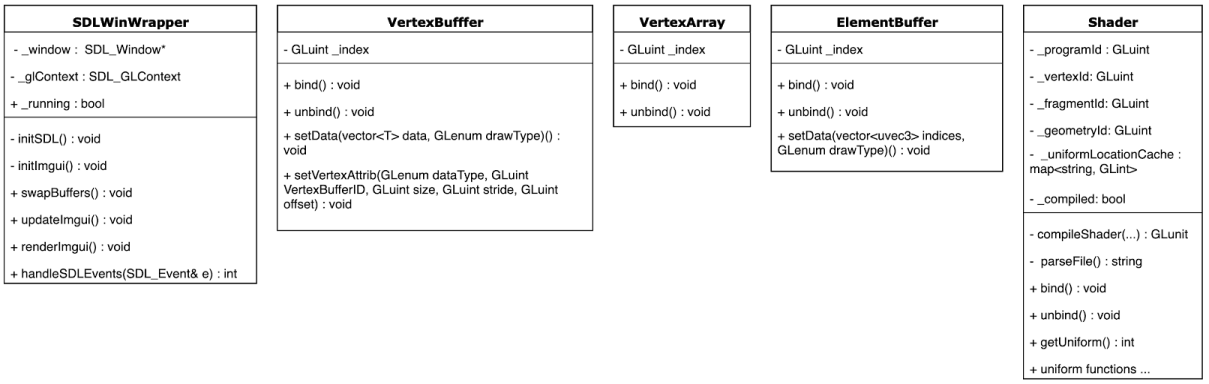


Illustration 4 : Classes openGL

2.2 Structure de données : stocker des cubes

Pour créer un tel logiciel à partir de zéro il a été nécessaire de choisir une structure de données adaptée.

Notre objectif est de stocker une couleur pour chaque cube ou voxel afin de dessiner notre scène. De nombreuses implémentations existent et nous allons vous présenter différentes implémentations possibles et les comparer afin de nous positionner sur l'une d'entre elles.

<u>Tableau statique</u>

De manière assez simple il est possible de considérer notre cube comme étant solide ou vide. Il est donc possible de représenter notre scène comme étant un tableau statique de

données (**std::array** par exemple) représentant la couleur de chaque cube avec une variable permettant de savoir s'il est vide ou non.

Le problème de cette approche est que l'on va envoyer de nombreuses données inutiles à notre carte graphique, en effet les cubes vides n'ont pas besoin d'être traités et il n'est donc pas nécessaire de les envoyer à la carte graphique.

L'avantage de cette méthode est que le temps d'accès pour une position donnée afin de créer, modifier, ou supprimer un cube est très rapide.

Tableau dynamique

Pour ne pas à avoir à stocker tous les cubes vides, il est possible d'utiliser un tableau dynamique (**std::vector** par exemple). Cependant, il va falloir stocker la position de chaque cube indépendamment sinon il est impossible de savoir où les dessiner dans l'espace. Par conséquent, cela fait plus de données à envoyer à la carte graphique.

Le gros inconvénient de cette méthode est que le **temps d'accès** à une position donnée est extrêmement **long**. En effet, le temps d'accès croit linéairement avec le nombre de cubes représentés dans la scène et pour éditer le cube en position (x,y,z) il est nécessaire de parcourir la liste de tous les cubes pour trouver le cube voulu, ou, dans le pire des cas, si celui-ci n'existe pas , parcourir toute la liste pour rien.

Tableau dynamique avec meshing

Une solution pour réduire le nombre de données envoyées à la carte graphique pourrait être d'utiliser une étape de "meshing". C'est à dire, effectuer un **calcul préalable** sur notre tableau dynamique afin de sélectionner uniquement les données utiles à envoyer à la carte graphique.

L'avantage de cette technique est que le **rendu** est **ensuite** bien **plus rapide**, mais le temps de meshing peut être très long, et **doit être réaliser à chaque modification du terrain** ce qui est problématique dans notre logiciel dont une des composante clé est l'édition et non seulement la visualisation.

Néanmoins, il reste toujour le probleme d'**accès** en **mémoire lent**.

Hashmap

Une solution est d'utiliser une hashmap. pour stocker la couleur de nos cubes en fonction de leur position. Une HashMap est une collection qui associe une clé (ici la position) à une valeur (ici notre couleur). Ainsi, il suffit d'utiliser une valeur clé (souvent un string ou un int) pour récupérer l'objet ou la valeur associée.

L'avantage est que l'utilisation d'une fonction de hachage de manière transparente dans l'implémentation améliore le temps d'accès.

L'inconvénient est que nos données ne sont plus stockées linéairement en mémoire et c'est un problème pour les envoyer au VBO. Une étape de conversion va être obligatoire (elle peut être combinée avec l'étape de meshing) afin d'obtenir un tableau continue en mémoire à envoyer à la carte graphique. De plus cette étape va être effectuée à chaque modification du terrain.

Octree

Une structure intéressante pour stocker nos cubes est une structure en octree. L'idée est de subdiviser un volume donné en 8 sous volumes et ce de manière récursive pour stocker nos cubes.

C'est une structure de données de type arbre dans laquelle chaque nœud compte huit fils. Chaque nœud occupe l'espace d'un cube et peut donc être subdivisé en huit parties égales.

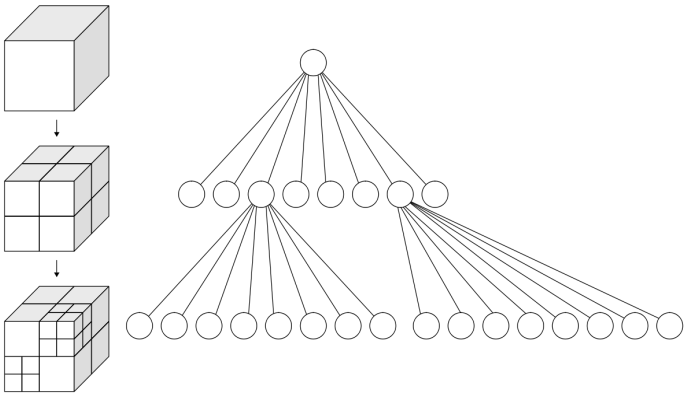


Illustration 5 : Structure en octree

Cette approche est intéressante car l'on peut envisager de stocker toute une zone de cubes identiques avec une seule information ou encore de faire du niveau de détail.

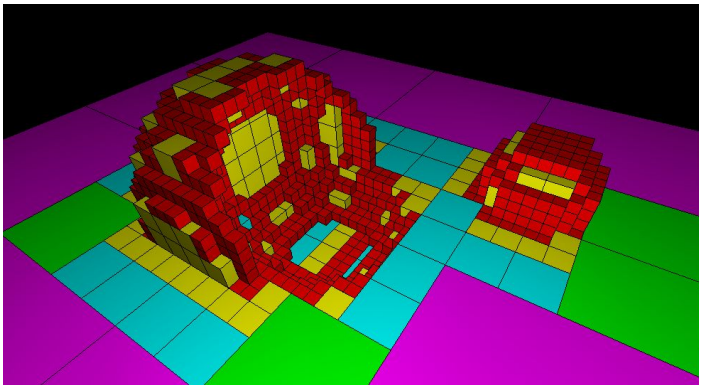


Illustration 6 : Exemple de niveau de détail avec une structure en octree

Cette structure permet également de faciliter les tests de collisions ou le rayCasting afin de ne pas tester toutes les collisions ou intersections possibles. Elle peut aussi être utilisé pour éliminer des objets en dehors du cône de vu d'une caméra dans le cadre d'un rendu 3D.

Illustration 7 : Exemple de rendu uniquement dans le "cône" d'une caméra

De plus les temps d'accès en mémoire sont plus rapides qu'avec une liste dynamique O(log(n)).

Bien qu'elle apporte des avantages elle vient aussi avec son lot d'inconvénients. L'accès est un peu plus rapide qu'avec une liste dynamique mais il reste plus lent qu'un tableau statique. De plus, l'**implémentation** est **plus complexe** et **plus gourmande en mémoire**. En effet il faut stocker des pointeurs vers chaque enfant dans chaque noeud et d'autres variables suivant l'implémentation.

Une étape de conversion est ici aussi nécessaire pour envoyer les données à notre carte graphique.

Choix final : Linked list Octree

Après réflexion le choix de l'octree semblait intéressant en prévision d'implémenter un raycasting pour notre curseur.

Néanmoins, ce problème de conversion en liste de données continues en mémoire était problématique pour une édition des cubes en temps réel sans latence lors de l'ajout ou suppression d'un cube.

C'est là que nous est venu l'idée d'utiliser à la fois la structure de donnée en octree et une liste dynamique.

std::vector<Cubedata> _data; (liste dynamique)
octree<unsigned int> octee;

L'idée est d'avoir une liste contenant les données (nos cubes) sans ordre particulier et un octree contenant uniquement des réels positifs (unsigned int) représentant l'index où est stockée la donnée dans la liste.

L'accès à un objet se fait par l'intermédiaire de l'octree ce qui nous permet de conserver ses avantages et l'on obtient un id via celui-ci qui nous permet enfin d'accéder à notre objet dans la liste.

L'ajout d'un cube se fait naturellement par l'ajout en tête de liste de notre cube (position et couleur) et de l'id dans l'Octree correspondant à cette case mémoire . De ce fait il n'y a plus d'étape de meshing à effectuer, nos **données** peuvent être **envoyées** directement à notre carte graphique.

En revanche, un problème est survenu lors de la suppression d'une donnée. En effet, supprimer un objet au milieu d'une liste dynamique peut être très lent si la liste est grande (complexité linéaire entre l'id de notre donnée et le nombre d'éléments derrière celle-ci)

Nous avons trouvé une idée permettant de contourner le problème mais plus gourmande en mémoire (c'est toujours une histoire de compromis).

Il s'agit d'une autre liste contenant des pointeurs vers nos index contenus dans l'octree. Cette liste va permettre d'effectuer un **swap** de données de complexité constante si notre objet n'est pas en tête de liste et de le supprimer avec un simple pop_back qui est de complexité constante lui aussi.

Prenons un exemple. Ci-dessous la liste de nos objets

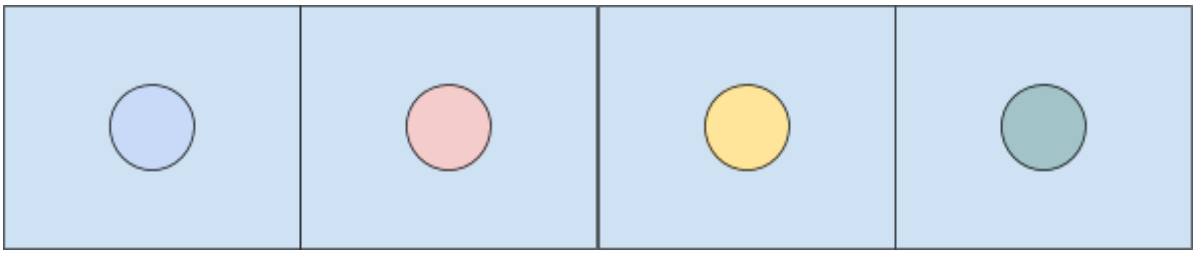


Illustration 8 : Exemple 2D "List Linked Octree", liste des objets

L'octree stock les ids correspondants à nos données (représenté sous la forme d'un quadtree par simplicité et la liste contient les pointeurs vers les ids de l'octree.

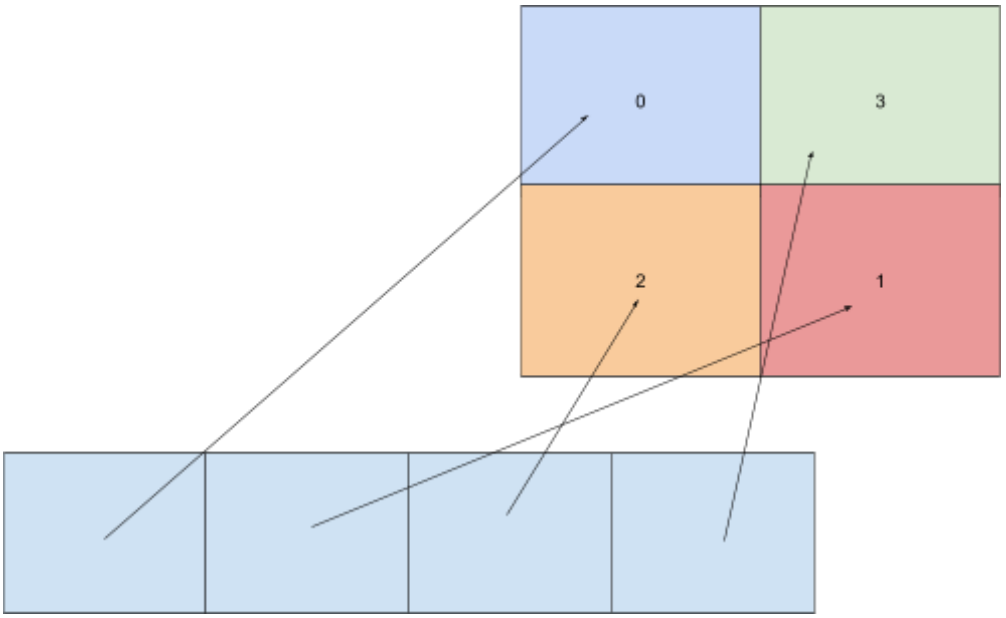


Illustration 9 : Exemple 2D "List Linked Octree", Octree stockant les ids des données et liste contenant les pointeurs

Lors de la suppression de l'objet rouge, un premier parcours dans l'octree permet de récupérer son id, ici 1 qui n'est pas en tête de liste.

Il est possible de connaître facilement l'id de l'objet en tête de liste qui égale à la taille-1. Par conséquent, nous pouvons faire facilement un swap des objets rouge et vert dans la liste.

Cependant, la position de l'octree ne pointera plus vers le bon objet. C'est là qu'intervient la seconde liste de pointeur qui va permettre de swap également les valeurs des ids correspondants dans l'octree. On obtient donc l'état suivant :

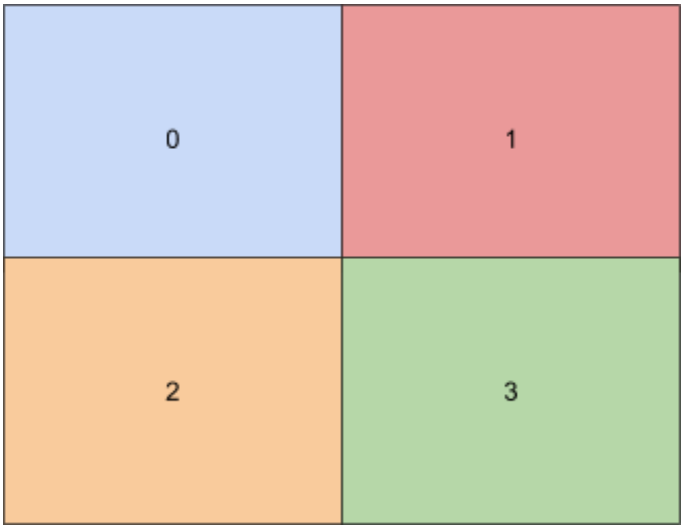


Illustration 10 : Exemple 2D "List Linked Octree", Octree stockant les ids des données après le swap

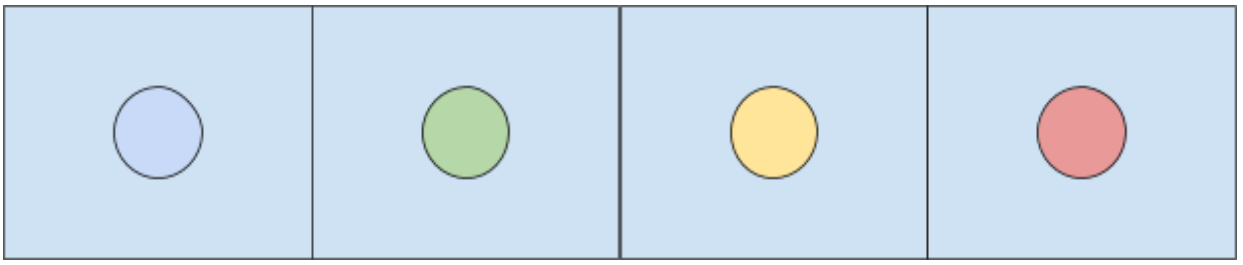


Illustration 11 : Exemple 2D "List Linked Octree", liste des objets après le swap

Il suffit alors de supprimer le dernier objet de la liste (complexité constante).

Il n'est pas non plus nécessaire de renvoyer toutes les données à la carte graphique, mais seulement la partie qui a changé, le cube vert. Cela peut se faire via la méthode glBufferSubData(..) le cube rouge étant ignoré lorsque que l'on précise le nombre de cubes à dessiner via la méthode glDrawArrays(GL_POINTS, 0, _cubes.vector().size());

Avec cette implémentation, l'ajout, la suppression ou l'édition d'un cube devient très rapide peu importe le nombre de données.

Néanmoins, elle apporte beaucoup de difficultés que nous évoquerons dans le bilan.

2.3 Rendu 3D

Une étape importante dans le rendu 3D va être de convertir nos cubes stockés en polygone afin de les dessiner. Cette étape peut être réalisée à plusieurs niveaux que nous allons détailler ici. Nous avons recensé trois implémentations possibles.

Meshing & vertex Buffer

L'idée est de générer, côté CPU, les différents polygones de rendu à partir de nos cubes et d'envoyer cette liste directement au GPU pour procéder aux calculs de lumières.

Il y a plusieurs façons de le faire et cela va être au développeur de trouver un compromis entre un meilleur mesh final avec un temps de calcul plus long ou un mesh un peu plus lourd mais bien plus rapide à calculer.

La version naïve est de générer 12 triangles par cube (2 par face). Le gros problème avec cette technique est que l'on envoie pleins de données inutiles (cubes invisibles car cachés par d'autres ou faces doublées entre deux cubes). Il a l'avantage d'être très rapide à calculer. Néanmoins, cela sera beaucoup trop lourd pour un grand nombre de cubes à afficher.

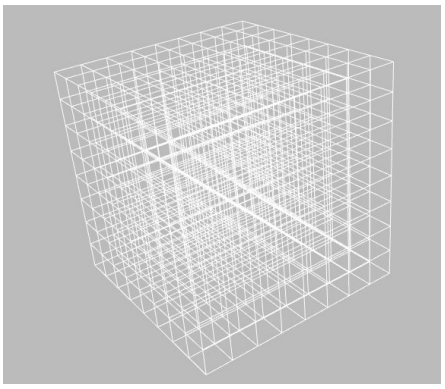


Illustration 12 : Rendu Wireframe avec la méthode “naïve”, 12 triangles par cube

Une autre façon de faire est d'utiliser la technique du “Face Culling”. L’idée est de calculer si un cube est caché (en regardant ses voisins) ou non et de générer uniquement les faces visibles.

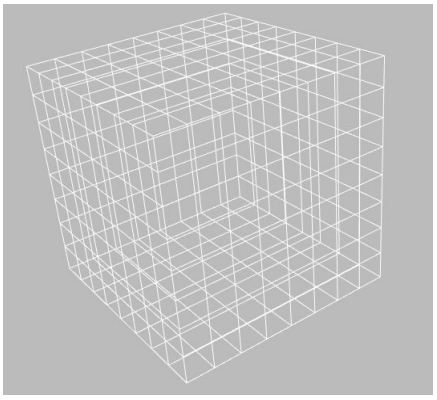


Illustration 13 : Rendu Wireframe avec la méthode du “Face Culling”

Une dernière technique dit de “Greedy meshing” peut aussi être utilisée pour combiner des surfaces planes et ainsi avoir un mesh final beaucoup plus léger. Cela peut faire une énorme différence comme ci-dessous où, pour une grille de 16x16x16, seulement **6** faces sont nécessaires au lieu de **1536** avec l’approche du Face Culling ou encore **24576** avec l’approche naïve mais elle prend du temps à être calculée ce qui est à prendre en compte.

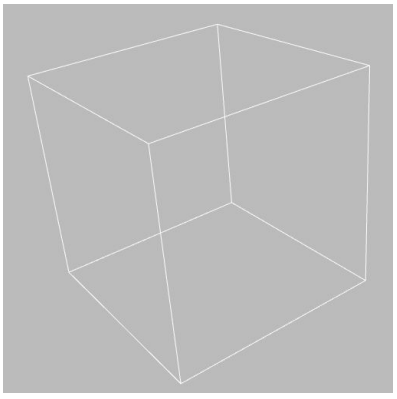


Illustration 14 : Rendu Wireframe avec la méthode du “Greedy meshing”

Instanciation

Une autre approche possible est de stocker le mesh déjà calculé d'un seul cube et d'utiliser un autre VBO contenant les positions des cubes pour redessiner les mêmes cubes à plusieurs endroits. Cette technique est très efficace quand il s'agit d'objet tel qu'un cube ici mais devient également problématique avec des objets plus lourds comme une sphère. Elle reste cependant très utilisée et efficace (c'est d'ailleurs celle qui nous a été présentée en TP).

Geometry shader

Une autre approche est possible avec l'utilisation de geometry shaders. L'idée est d'envoyer uniquement les positions des cubes à la carte graphique et de générer les faces à afficher ou non à l'aide du GPU.

L'avantage de cette technique est que l'on envoie peu de données à la carte graphique. En revanche, le calcul des faces à générer doit être effectué à chaque frame ce qui peut être un problème.

Cependant, la génération des faces va profiter de la puissance de notre GPU.

Le combat de performance entre la technique d'instanciation ou de geometry shader et très répandu aujourd'hui et chaque méthode a ses avantages et ses inconvénients.

Par curiosité, nous avons opté pour la technique des geometry shaders afin de voir son potentiel.

FaceMask culling

Nous avons trouvé une solution pour effectuer le "Face Culling" directement lors de l'ajout d'un cube à la scène. En plus de la position et la couleur, nous stockons un masque binaire (tenant sur un octet : uint8_t) permettant de savoir si le cube est collé à un autre cube.

Cette étape peut être faite directement lors de l'ajout d'un cube à la scène, en regardant les voisins. Nous avons choisi un ordre de parcours des faces arbitraire fixé : *haut, bas, gauche, droite, devant et derrière*. Lors de la suppression d'un cube il suffit de mettre à jour les masques des cubes voisins pour la face en question.

Notre masque binaire contient un 0 au bit correspondant à la direction si un cube est collé au cube actuel dans cette direction. Par exemple, pour le cube en beige ci dessous le mask sera le suivant : **010010.** En effet, les seules directions dans lesquelles le cube n'a pas de voisins sont les directions *bas* et *devant* qui correspondent respectivement au bit 5 et 2 en partant de droite.

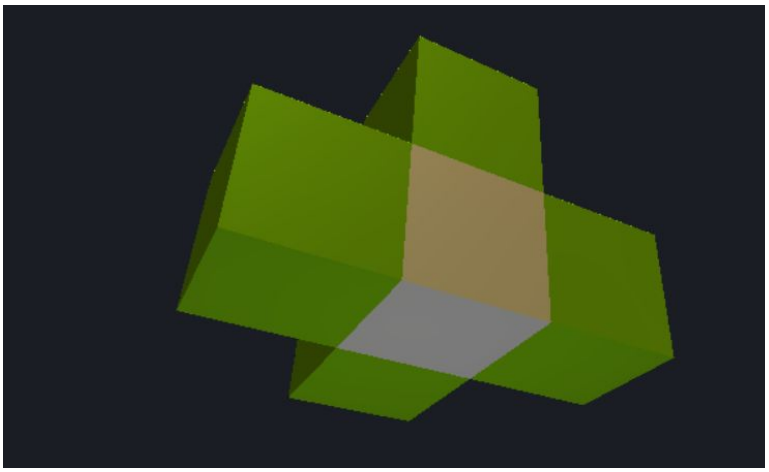


Illustration 15 : Exemple “FaceMask culling”

Ce masque binaire va permettre au geometry shader de générer uniquement les faces visibles (en effet il est inutile de générer les deux faces respectives de deux cubes côte à côte).

```
// left
if( (faceMask & 0x04) != 0) {
        // generate left face
}
```

On peut voir ci-dessous une vue depuis l'intérieur des cubes où l’on s'aperçoit que les faces intérieures ne sont pas dessinées.

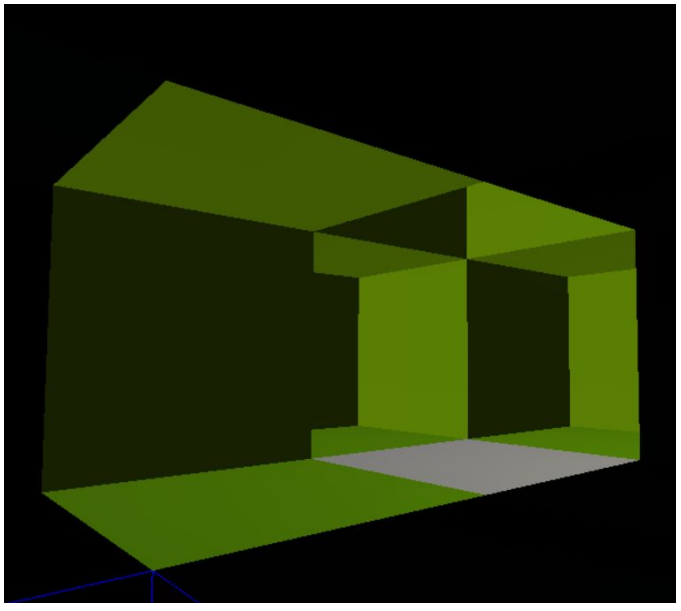


Illustration 16 : Exemple “FaceMask culling”, faces intérieures

2.4 Caméra

Nous avons choisi de réaliser une caméra “Free Fly” avec des angles d’Euler, à la manière d’un jeu vidéo pour permettre à toto de bouger n'importe où dans la scène.

La caméra est définie par une position et trois angles d’Euler (en anglais yaw, pitch, rool) qui permettent d'orienter la caméra dans l’espace.

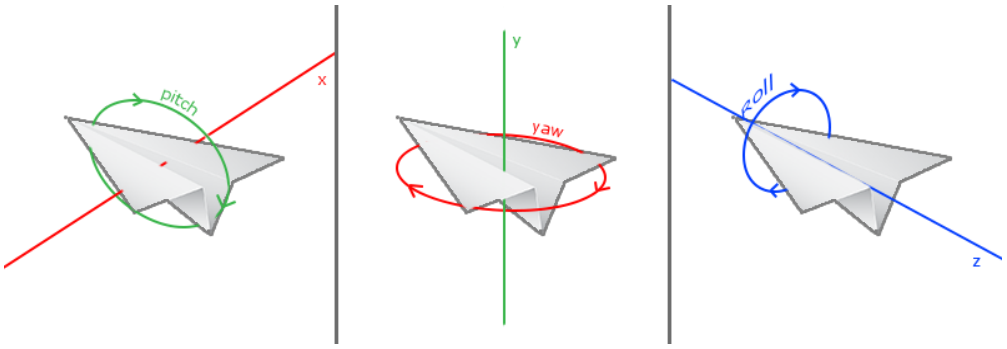


Illustration 17 : Schéma explicatif des angles d’Euler

Pour la caméra nous allons seulement nous préoccuper du **yaw** et du **pitch** qui vont permettre, après un peu de calcul trigonométrique, d’obtenir un vecteur de direction de la manière suivante:

```
direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
direction.y = sin(glm::radians(pitch));
direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
```

Nous allons également définir le vecteur pointant vers le haut de notre monde (axe y) pour permettre à la caméra de rester droite (le roll est implicitement défini nul grâce à ce vecteur).

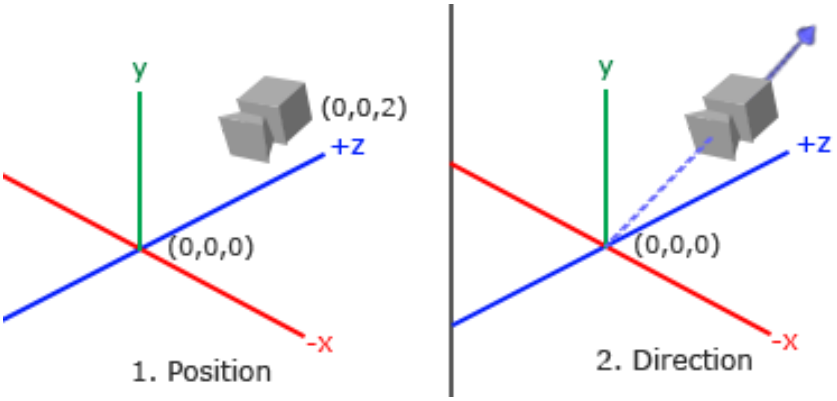
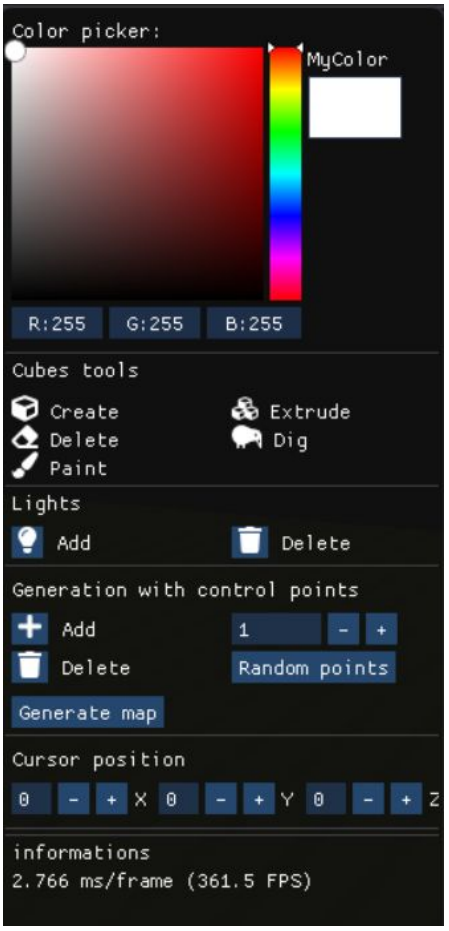


Illustration 18 : Schéma explicatif de la position et de la direction d’une caméra

A l’aide de ces trois informations et de quelques produits vectoriels il va être possible de déterminer la matrice de projection de notre caméra.

2.5 Interface et outils d'édition



Pour l'interface utilisateur, nous avons utilisé la librairie ImGui qui propose un large choix de widgets. De plus, ImGui a l'avantage de gérer ses événements ce qui représente un gain considérable. Par ailleurs, nous avons intégré des icons "font Awesome" afin d'améliorer l'expérience utilisateur.

La fenêtre d'édition se trouve sur la droite de l'interface et fait toute la hauteur de la fenêtre de l'application. Elle est divisée en plusieurs sections correspondants aux différentes fonctionnalités d'éditions.

Ayant fait le choix de proposer des cubes de différentes couleurs, un color picker permet à l'utilisateur de choisir parmi toutes les couleurs RGB. Le color picker influence l'outil "create", l'outil "paint" ainsi que l'outil "add light".

Concernant la section la section "Cubes tools", il s'agit d'un sélecteur permettant de changer la stratégie des outils. En effet, nous avons utilisé la classe **std::function** dans le cadre d'un pattern stratégie pour définir l'action à effectuer en fonction de l'outil choisi.

La classe template **std::function** permet de manipuler des fonctions, fonctions anonymes (lambda), bind (application partielle de fonction) ou encore des foncteurs (objet-fonction) comme des variables et cela nous paraissait un choix pertinent ici pour assigner une nouvelle fonction à notre outil lors d'un changement depuis le menu ImGui. Une fois la couleur et l'outil sélectionné, l'utilisateur n'a plus qu'à déplacer le curseur et appuyer sur la touche A du clavier pour éditer les cubes de la scène.

Tout ce qui concerne l'interface et les outils d'éditions se trouvent dans le répertoire **/gui**, avec le namespace du même nom. La création de l'interface et les fonctions d'éditions sont dans la classe **Menu** qui délègue la gestion de la stratégie à la classe **Tool**. **Menu** interagit avec l'**App** via la classe **AppSettings** qui prend des pointeurs vers les données nécessaires à l'utilisation de l'interface.

ImGui permet également d'ajouter une barre de navigation avec plusieurs onglets en haut de la fenêtre.

Illustration 20 : Barre de navigation

Nous retrouvons donc plusieurs options générales ou avancées notamment le reset de la scène, les options du soleil, le paramétrage des fonctions radiales ou encore l'onglet Help. Ce dernier résume toutes les actions possibles au clavier, à la souris ou via l'interface d'édition.

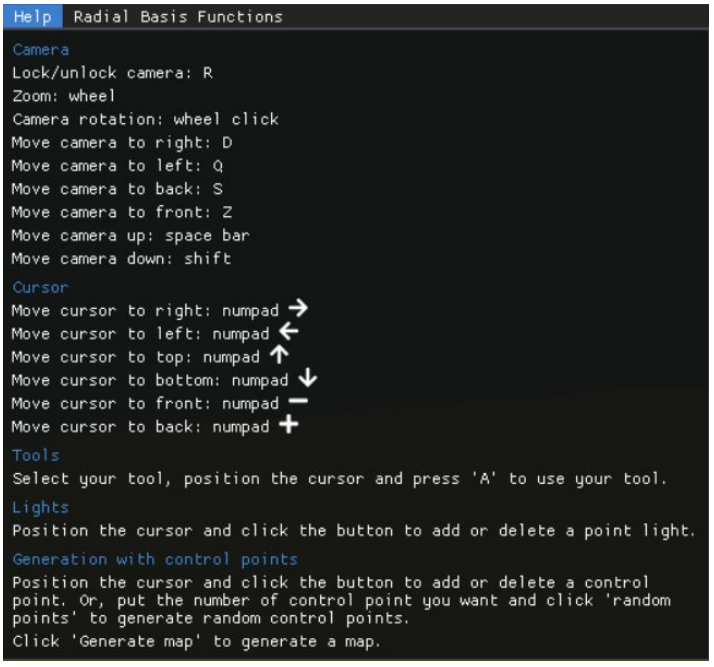


Illustration 21 : Onglet "Help" de la barre de navigation

2.6 Lumières et shaders

Pour gérer les lumières dans notre scène, nous avons utilisé deux techniques : une **lumière directionnelle** pour le soleil, et plusieurs lumières ponctuelles pour décorer notre scène.

Nos informations sont passées via des variables uniform à nos différents shaders, c'est dans ceux-ci que les calculs de lumière pour chaque pixel (ou plus précisément fragment et ensuite pixel) de notre écran seront calculés.

```
uniform vec3 sunDir;
uniform vec3 sunColor;
uniform float dayMode;

uniform int nbOfPointsLight;
struct PointLight {
    vec3 pos;
    float constant;
    float linear;
    float quadratic;
    vec3 ambientColor;
    vec3 diffuseColor;
};

uniform PointLight pointsLights[MAX_LIGHTS];
```

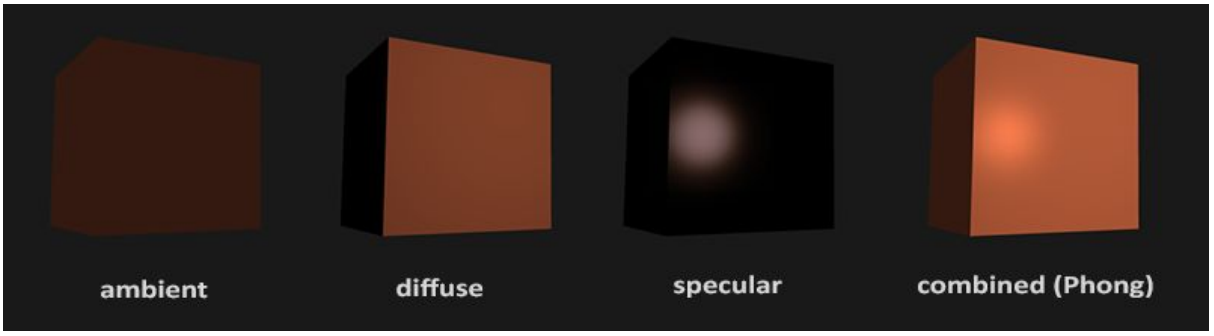


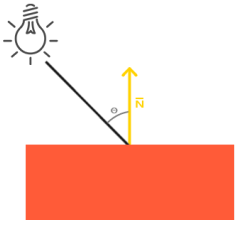
Illustration 22 : Exemple de différents types de lumières

Nous avons choisi d'implémenter un modèle simple pour nos lumière : le modèle de Phong.

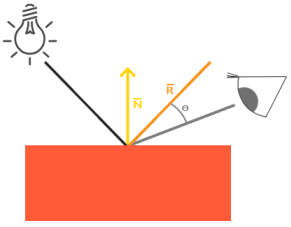
Une première composante est la lumière ambiante, elle représente la lumière globale ambiante de la scène (comme son nom l'indique). Cela permet de s'affranchir de calculs beaucoup trop coûteux et complexes en ressources de réflexions de lumières.

Donc, l'astuce est de simplement imiter cette lumière afin de ne pas apparaître complètement noire.

La composante diffuse de la lumière permet de rendre compte des reliefs et calculer à l'aide de la normale de la surface, l'intensité de celle-ci. Une lumière arrivant de biais éclaire moins qu'une lumière de face.



Enfin une composante spéculaire permet de rendre compte de la lumière réfléchie par l'objet et de son intensité en fonction de la position d'observation.



Concernant nos lumières ponctuelles deux calculs viennent s'ajouter:

Un calcul d'atténuation de la lumière (à l'aide de 3 coefficient) en fonction de la distance via la formule :

$$\frac{1}{K_c + K_l.d + K_q.d^2}$$

En effet, plus un objet et loin moins il est éclairé par notre source ponctuelle ce qui est assez instinctif .

Nous voulions aussi réaliser une lumière plus “cartoon” en utilisant un coefficient (toonShading) permettant de créer des paliers pour l’intensité de la lumière diffuse ce qui crée cet effet craqué comme nous pouvons le voir sur l’image ci-dessous.

```
const int toonShadingLvl = 8;
float diffuseIntensity = max(dot(normal, lightDir), 0.0);
diffuseIntensity = floor(diffuseIntensity * toonShadingLvl) / float(toonShadingLvl);
```

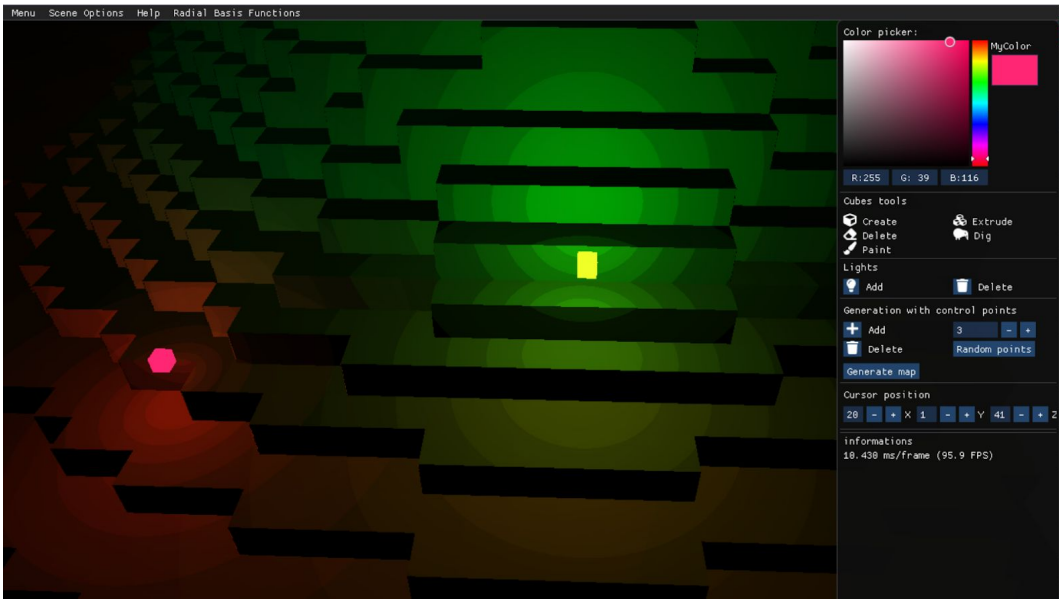


Illustration 25 : Exemple de toon shading

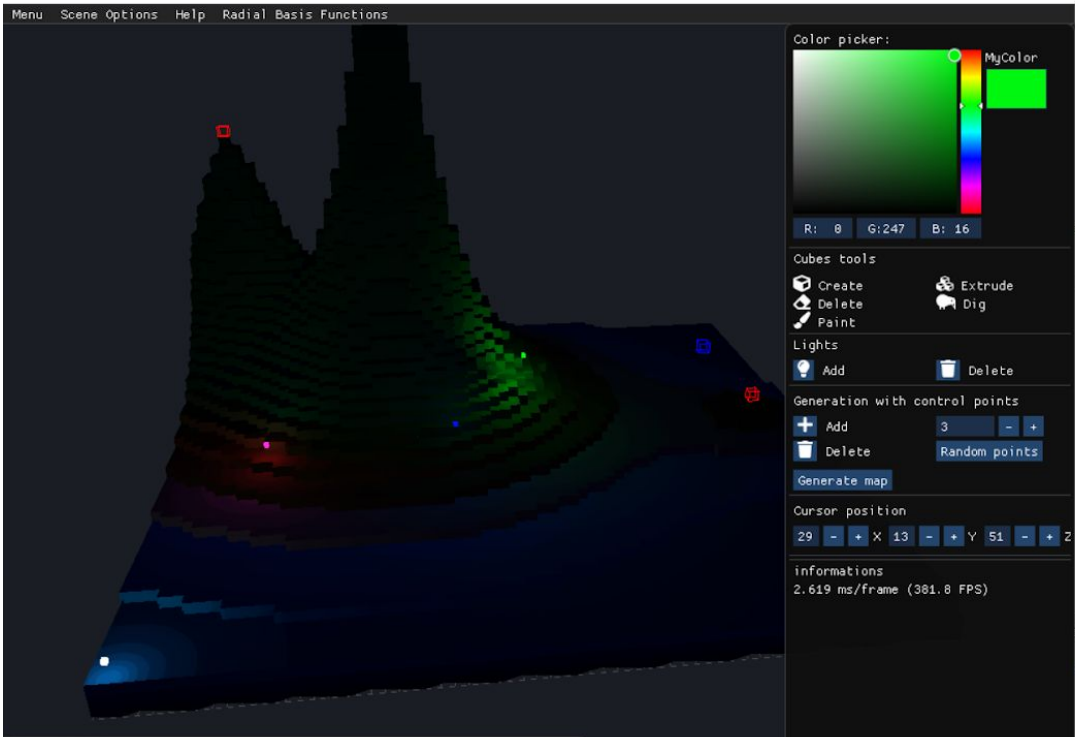


Illustration 26 : Exemple d’une scène de nuit avec plusieurs points de lumière

Un onglet du menu permet de changer la lumière ambiante pour passer d'une ambiance jour à une ambiance nuit ainsi que d'animer notre soleil pour simuler une journée (avec une rotation de la direction de la lumière).

Illustration 27 : Capture d'écran du menu option de la scène

2.7 Génération procédurale : Fonction à base radiale (Radial Basis Function)

La génération procédurale de notre terrain était une des fonctionnalité principale de ce projet. Nous avons utilisé les outils de la STL (Standard Template Library) pour générer des points de contrôles aléatoirement dans notre scène. Ces points sont utilisés pour définir une hauteur à atteindre. L'objectif est d'interpoler la hauteur des points intermédiaires .

On définit, avec des fonctions radiales (dépendante de la distance), l'influence de chaque point de contrôle sur les positions environnantes .

Une pensée rapide pourrait être de faire un produit de la hauteur (ui) en fonction de la distance d'un point de contrôle (xc) avec notre position x quelconque. Plus la distance est grande moins le point x est influencé par les points de contrôles (fonction radiale décroissante).

$$g(x) = \sum_{i=1}^{k} u_i.\phi(|x - x_i|)$$

Cependant cette approche fonctionne bien avec uniquement un point de contrôle.

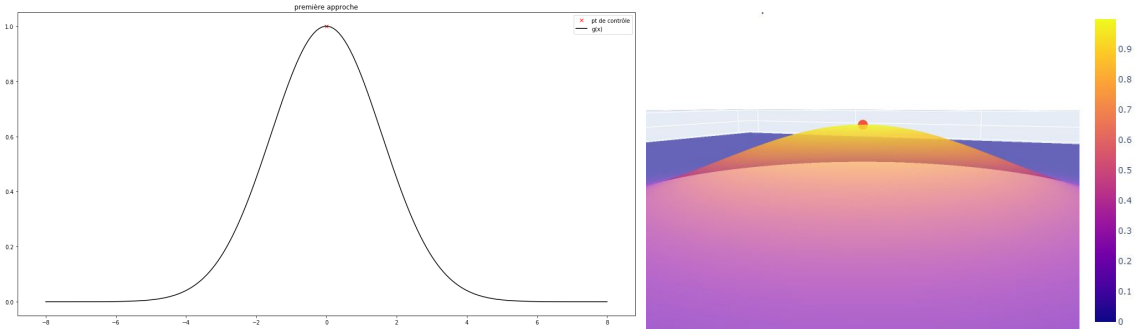


Illustration 28 : Première approche avec 1 point en 2D à gauche, en 3D à droite

On peut voir ci-dessous en 2D que cela ne fonctionne plus avec deux points de contrôle, la courbe ne passe plus par nos points (Or c'est ce que l'on souhaite)

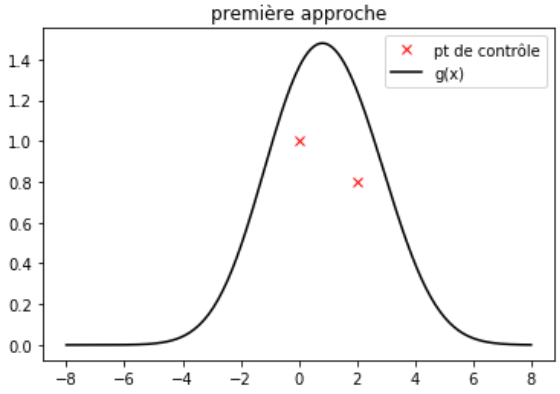


Illustration 29 : Première approche avec 2 point en 2D

Pour trouver une bonne interpolation passant par plusieurs points de contrôle il faut utiliser les maths. On définit la fonction g(x) telle que :

$$g(x) = \sum_{i=1}^{k} \omega_i . \phi(|x - x_i|)$$

L'idée est de conserver notre première approche et déterminer des ωi inconnus de manière à conserver la condition suivante : (où ui est la hauteur correspondante au point de contrôle xi)

$$(\forall i \in [1, k]\ g(x_i) = u_i)$$

Grâce à nos points de contrôle on connaît k contraintes sur les ωi :

$$\sum_{i=1}^{k} \omega_i . \phi(|x_m - x_i|) = u_m \quad \forall m \in 1 \ldots k$$

Ce qui va nous permettre de construire un système linéaire (matricielle) et trouver nos ωi à l'aide de la librairie Eigen :

```
Eigen::VectorXf rbf::computeOmega(const std::vector<Eigen::Vector2f> &controlePts,
const Eigen::VectorXf &controlePtsValues , std::function< float(float) > radialFunction) {
    Eigen::MatrixXf A = Eigen::MatrixXf::Zero(controlePts.size(),controlePts.size());
    // build A with controlePts & radialFunction
    for (int j = 0; j < A.cols(); j++) {
        for (int i = 0; i < A.rows(); i++) {
            A(i, j) = radialFunction(dist(controlePts[i], controlePts[j]));
        }
    }
    // solve system using Eigen
    Eigen::ColPivHouseholderQR<Eigen::MatrixXf> dec(A);
    return dec.solve(controlePtsValues);
}
```

Il suffit ensuite d'appliquer la fonction à nos positions intermédiaires (evaluationPts)

```
Eigen::VectorXf interpolateValues = Eigen::VectorXf::Zero(evaluationPts.size());
    for (size_t i = 0; i < evaluationPts.size(); i++) {
        for (size_t j = 0; j < controlsPtsValues.size();j++) {
            interpolateValues(i) += w(j) * radialFunction(dist(controlsPtsPos[j], evaluationPts[i]));
        }
    }
    return interpolateValues;
```

Cela nous donne bien le résultat escompté (nos point de contrôle représentés ci-dessous en rouge)

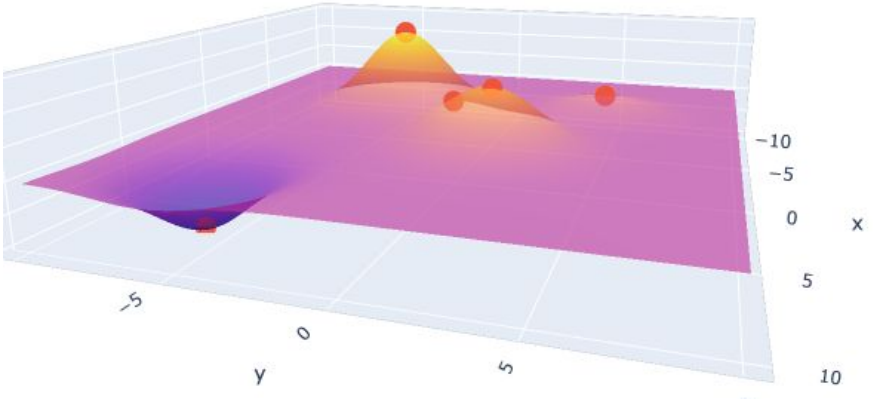


Illustration 30 : Résultat de radial basis function

Depuis notre application il est possible de définir différents coefficients pour manipuler les fonctions radiales et générer différents terrains :

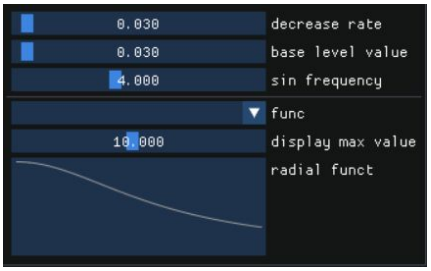


Illustration 31 : Capture d'écran du menu radial basis function

Une fonction d'interpolation de couleurs va se charger d'attribuer une couleur aux cubes en fonction de leur hauteur suivant un gradient de couleur (https://cssgradient.io/)

Illustration 32 : Dégradé utilisé pour définir la couleur des cubes en fonction de la hauteur

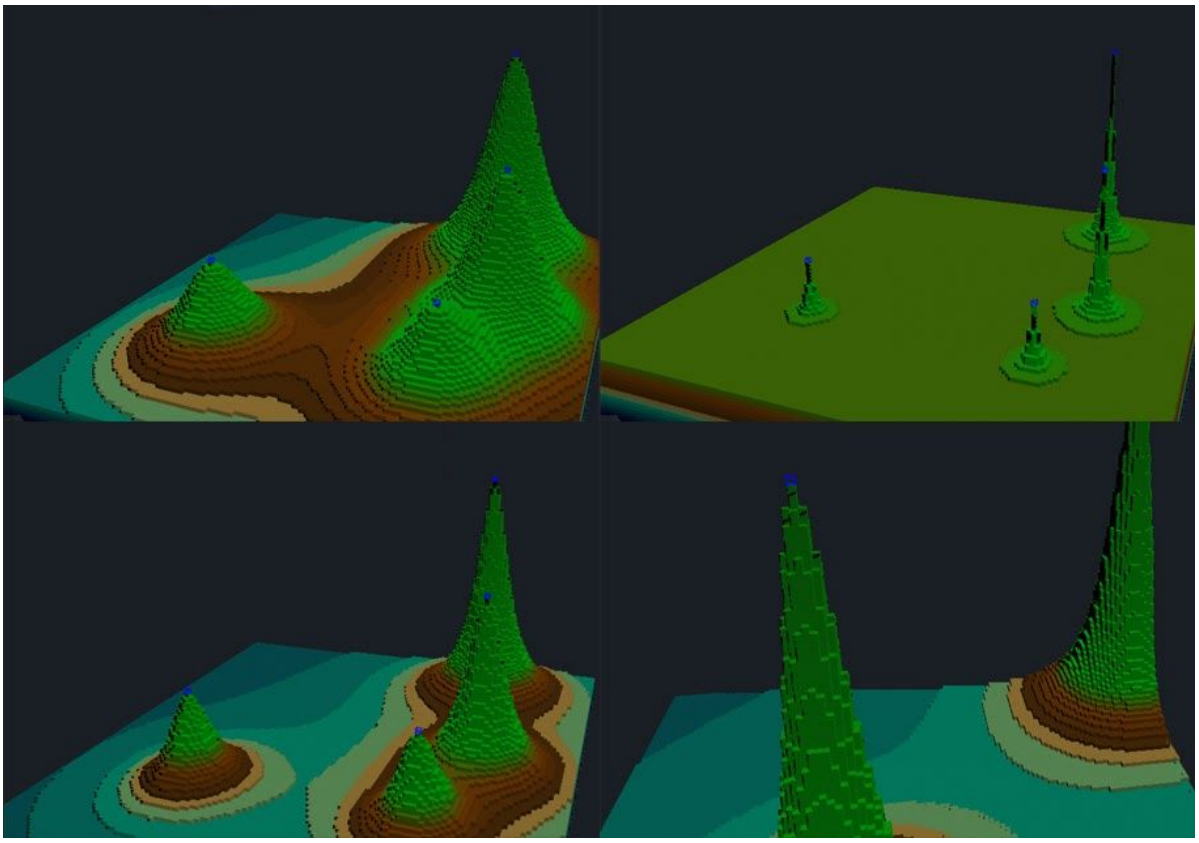
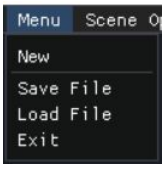


Illustration 33 : Résultat de génération procédurale de terrain en utilisant des fonctions radiales différentes, points de contrôles en bleu

2.8 Save and load

Pour pouvoir sauvegarder notre scène, nous avons utilisé des outils filestream de la STL (**std::ofstream**) une fonction écrivant les données de chaque cube de la scène dans un fichier “save” en binaire .




```
for(const CubeVertex &cv : _data) {
        file.write((char*)& cv.pos.r, uIntSize); // Position
        file.write((char*)& cv.pos.g, uIntSize);
        file.write((char*)& cv.pos.b, uIntSize);
        file.write((char*)& cv.color.x, floatSize); // Colour
        file.write((char*)& cv.color.y, floatSize);
        file.write((char*)& cv.color.z, floatSize);
    }
```

Pour charger le terrain il suffit de faire l'opération inverse pour ajouter chaque cube à un chunk préalablement vidé.

```
reset();
...
unsigned int* posBuffer = new unsigned int[3] { 0 };
float* colorBuffer = new float[3] { 0.0f };

    while(file.peek() != EOF) {
        file.read((char*)posBuffer, 3 * sizeof(unsigned int));
        file.read((char*)colorBuffer, 3 *  sizeof(float));
          ...
 setColor(glm::uvec3(unsigned int(posBuffer[0]), unsigned int(posBuffer[1]),unsigned int(posBuffer[2])),
glm::vec3(colorBuffer[0], colorBuffer[1], colorBuffer[2]));
  }
```

3. BILAN GÉNÉRAL

3.1 Difficultés rencontrées

Ce projet, complet et dense, nous a fait rencontrer quelques difficultés. Dans un premier temps, Enguerrand, étant arrivé dans la formation en deuxième année, ne connaissait pas openGL et a dû apprendre à l'utiliser. Néanmoins, il a réussi à se mettre à niveau rapidement et ce fut par la suite à Amandine d'apprendre à travailler sur un niveau supérieur à ses compétences actuelles.

Ensuite, nous devions avoir une vision d'ensemble du projet. Cela n'a pas été facile de trouver comment organiser le projet, savoir quelles classes faire, comment les lier, trouver une structure de données adaptée, répartir les classes dans des dossiers et des namespace. Nous pensons avoir réussi à mettre en place quelque chose de fonctionnelle et adaptée en fonction du cahier des charges et des contraintes de temps.

Néanmoins, le temps imparti et la gestion des heures de travail sur ce projet en parallèle des cours et des autres projets peuvent être noté en difficulté. En effet, nous n'avons pas eu le temps d'implémenter toutes les fonctionnalités souhaitées. Par exemple, nous avions choisi la structure de donnée en Octree dans l'optique d'améliorer le curseur en ray casting. Mais finalement, nous n'avons pas eu le temps d'ajouter cette fonctionnalité.

En effet, l'implémentation de la structure en Octree, afin d'avoir beaucoup de données sans perdre en fluidité, fut plus compliqué à mettre en place que nous le pensions. Par exemple, la gestion des données vide (une position non attribué) nous a posé problème et nous avons donc utilisé des pointeurs pour y remédier. Cette solution n'est cependant pas idéale. De plus, l'accès aux différents objets de l'application depuis la classe menu fut également assez problématique. Nous avons créé une classe intermédiaire (appSettings) qui permet au menu de manipuler les objets via des pointeurs.

Cette solution n'est pas idéale et l'utilisation de forward déclaration et de références sera surement plus adapté mais nous n'avons pas eu le temps de le changer. Malgré cette architecture complexe et ce choix finalement moins pertinent, cette structure est très intéressante pour beaucoup d'applications et nous avons beaucoup appris.

Nous avons rencontré quelques difficultés avec les geometry shaders. En effet, nous n'en avions pas vu en TP et nous devions donc apprendre à les utiliser. Nous avons vu que le geometry shader est un outil très puissant qui permet de modifier la géométrie. Par exemple, il est possible d'appliquer des effets comme des smooth ou des tremblements.

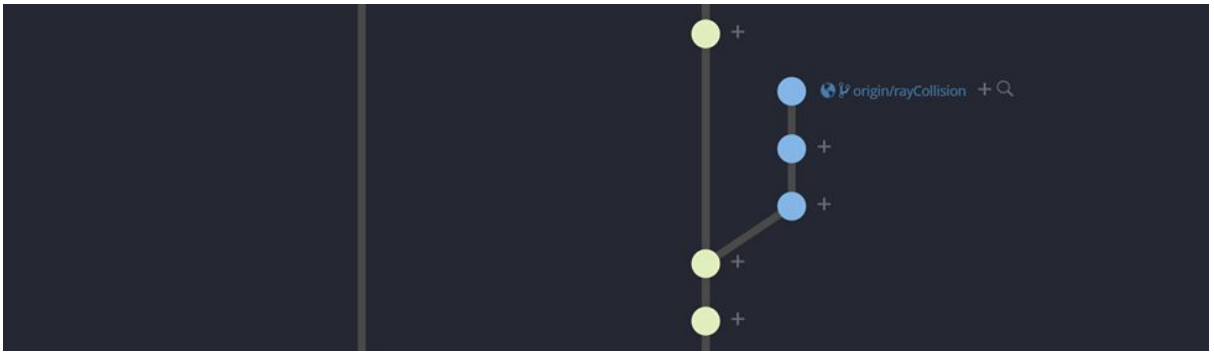


Illustration 35 : Capture d'écran de la branche du ray casting que nous n'avons pas eu le temps de finir

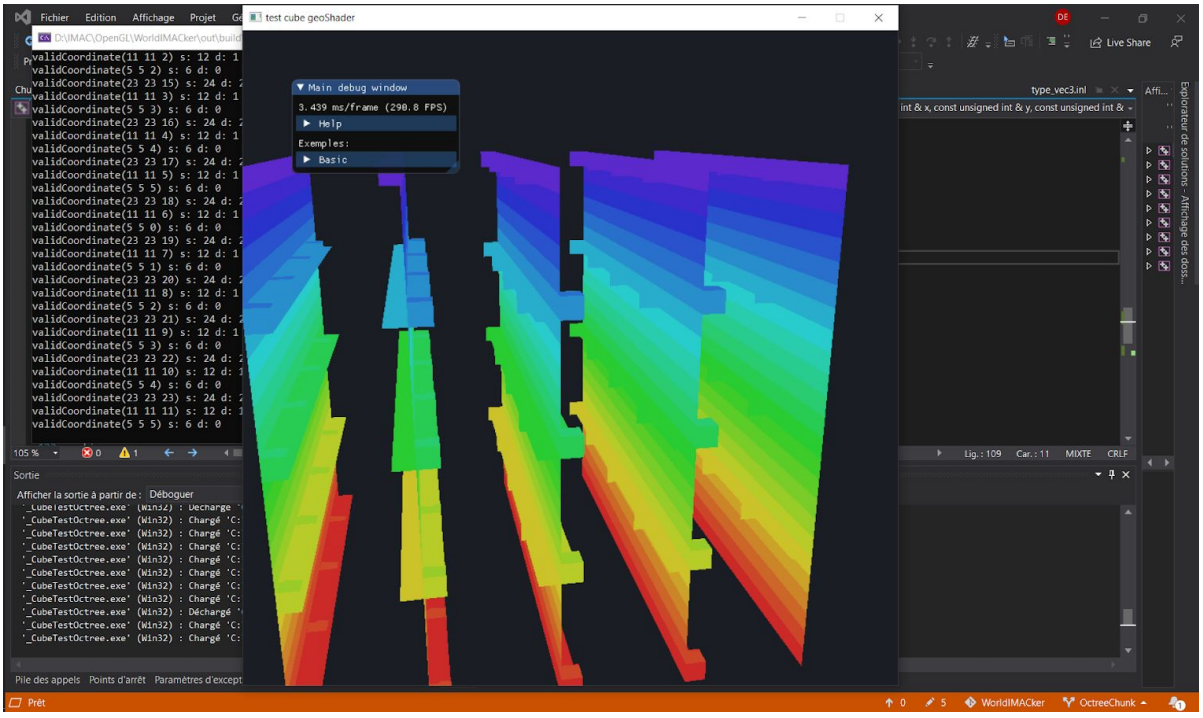


Illustration 36 : Capture d'écran d'un bug avec l'Octree

3.2 Perspectives d'améliorations

Nous avons noté plusieurs points qui pourraient être améliorés dans une version future. Dans un premier temps, nous pourrions exploiter davantage la structure en Octree en implémentant un niveau détail, que nous avons évoqué précédemment. C'est-à-dire, stocker la couleur de toute une zone et ne dessiner qu'un seul rectangle de cette couleur plutôt que plusieurs petits.

De plus, nous pourrions ajouter la technique de frustum culling avec le cône de caméra, c'est-à-dire calculer toutes les zones visibles par la caméra mais ne pas dessiner celles qui sont hors du cône de la caméra.

Nous pourrions également simplifier l'implémentation de l'Octree en utilisant moins de pointeurs, ou encore améliorer la recherche de voisins avec un algorithme intelligent afin de ne pas parcourir l'Octree à partir de la racine.

D'autre part, notre projet possède des chunk. Le principe d'un chunk est de diviser le monde en plusieurs partitions afin de charger ou décharger des zones en fonction de où on se trouve et de la distance de la caméra. C'est ce que nous voulions faire dans un premier temps mais les contraintes de délais ne nous ont permise de ne faire qu'une seule zone. Nous pourrions donc implémenter ce principe afin de décharger les blocs à partir d'une certaine distance et de ne pas les afficher.

Par ailleurs, nous pourrions terminer l'amélioration du curseur, le ray casting.

Nous pourrions également utiliser les frameBuffer pour générer des ombres (shadow mapping) ou encore améliorer la lumière avec de l'occlusion ambiante.

Nous pourrions aussi ajouter, en plus des radiales basis functions, du bruit de perlin qui nous permettrait d'avoir des creux dans la map.

Les geometry shaders nous permettrait de proposer des effets à l'utilisateur comme un smooth.

4. BILAN PERSONNEL

4.1 Enguerrand De Smet

Ce projet était une grande première pour moi dans le monde d'openGl et de la programmation bas niveau en c++.

J'ai appris énormément de chose à travers ce projet, que ce soit dans le fonctionnement du pipeline graphique ou même en terme d'organisation et de travail en équipe.

Ce projet fut très dense pour nous deux, malgré une légère différence de niveau nous avons pris le temps ensemble de discuter et de réfléchir aux différentes fonctionnalités à implémenter. L'utilisation de git fut aussi très utile pour se répartir le travail et travailler à distance.

J'ai pu expérimenté l'implémentation d'une nouvelle structure de données un peu moins classique qu'un simple tableau à savoir l'octree. Cela a été un réel défi pour moi et je pense en avoir tiré une grande leçon que cette citation de Donald Knuth illustre très bien **"*Premature optimization is the root of all evil*".** En effet, j'ai passé beaucoup de temps à réfléchir sur ce problème, vouloir optimiser et minimiser les échanges de données avec les buffer de la carte graphique et me suis créé des problèmes où il n'y en avait parfois pas (utilisation du code morton pour le parcours dans l'octree).

Je suis content du résultat bien que très loin d'être parfait et un peu oldSchool (utilisation abusive de pointeurs) mais j'ai beaucoup appris sur les structures de données lors de mes recherches et je referai surement mieux dans le futur.

J'aurai pu à la place me concentrer sur d'autres fonctionnalités, les lumières ou une sauvegarder un peu plus adaptée. L'utilisation de geometry shaders fut aussi très intéressante. Je me suis rendu compte des possibilités apportées par les shaders de manière générale et j'ai bien envie de creuser le sujet ainsi que les compute shaders qui me semble aussi très intéressant d'un point de vu des performances.

Avec le temps imparti et les autres projets et révisions je n'ai pas pu ajouter certaines fonctionnalités que j'aurai aimé implémenter (le raycasting, ou un système de Level of detail), mais les recherches effectuées lors du projet ne sont pas vaines et me seront surement utiles dans mes projets futurs.

J'ai beaucoup apprécié travailler en équipe avec Amandine sur ce projet, bien que légèrement moins à l'aise en programmation, elle était volontaire et cela m'a aussi apporté beaucoup, et m'a permis de sortir de ma zone de confort, de collaborer et d'échanger pour trouver des solutions ensemble.

4.2 Amandine Kohlmuller

J'apprécie beaucoup la programmation, surtout lorsque l'on peut visualiser graphiquement ce que l'on fait. Néanmoins, je ne suis pas spécialisée dans ce domaine et je ne compte pas en faire mon métier. Par conséquent, il était difficile de m'adapter et de m'aligner à mon binôme qui, contrairement à moi, résonne aisément pour trouver des solutions aux problématiques d'un tel projet. Mais Enguerrand a pris en compte l'écart de niveau afin de m'intégrer au projet de manière adéquate.

J'ai davantage travaillé sur le curseur pour se déplacer dans la scène, l'interface avec ImGui et l'implémentation de la stratégie pour les outils d'éditions. J'ai également commencé l'implémentation de la génération procédurale et des fonctions radiales avec Eigen. De manière générale, Enguerrand était vraiment très rapide pour moi. Quelque chose de très simple pour lui pouvait représenter un réel défi pour moi. Dans ce genre de situation la personne avec le plus de difficultés peut facilement se retrouver écartée du projet et se laisser aller en attendant que les choses se fassent. Ce ne fut pas mon cas et et je suis heureuse d'avoir réussi à m'intégrer et m'impliquer autant que possible dans ce projet très enrichissant. Enguerrand prenait le temps de m'expliquer et nous réfléchissions ensemble aux différentes solutions d'implémentation. Je n'ai certes pas tout compris dans les détails, mais j'étais capable de réagir et de réfléchir avec lui sur les différents sujets.

Par ailleurs, le court délai de réalisation laisse derrière quelques frustrations. Par exemple, lors des TP de synthèse d'images, j'ai beaucoup apprécié travailler sur les shaders et toutes les possibilités qu'ils offrent. Je me suis amusée sur shadertoy en essayant de créer des textures animées et en passant beaucoup de temps sur le site the book of shaders. Malheureusement, avec l'approche de la deadline, les autres fonctionnalités du cahier des charge et les autres projets et révisions, je n'ai pas trouvé le temps de continuer mes explorations dans le monde des shaders afin d'en intégrer dans notre projet.

Toutefois, ce projet m'a apporté beaucoup de connaissances techniques, notamment sur l'abstraction d'openGL, les différentes structures de données ou encore l'application concrète de design patterns. J'ai également acquis de nouvelles compétences en communication. En effet, il était parfois très difficile pour moi de suivre le rythme d'un programmeur avec davantage de connaissances et de compétences. Néanmoins, ce projet était un assez bon reflet de ce à quoi j'aspire professionnellement, c'est-à-dire travailler avec des spécialistes sans en être une mais être capable de communiquer avec eux et chercher des solutions en ayant un aperçu global du projet.