

# C++ - Synthèse d'image - Maths info

## Projet : World IMaker

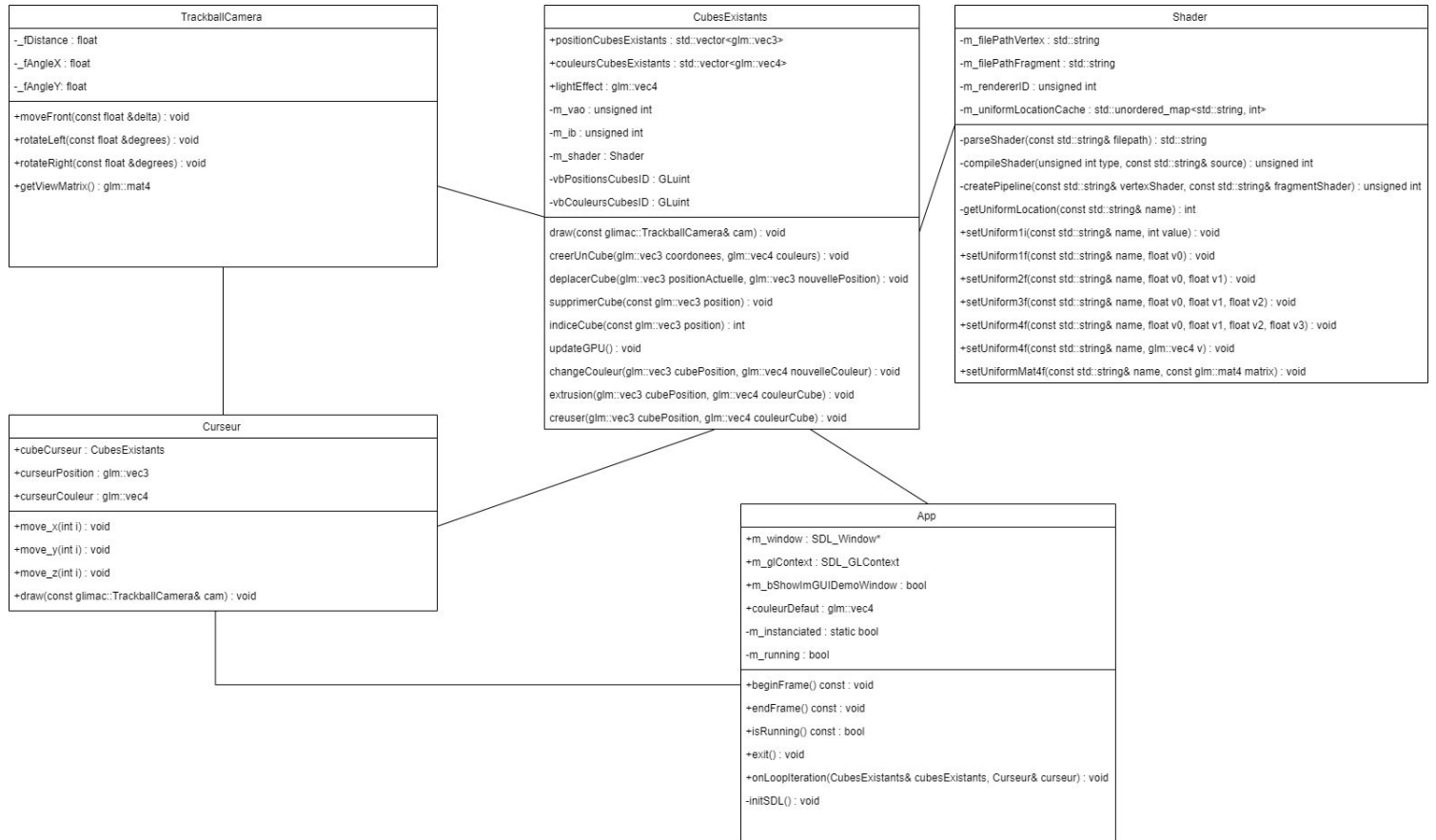


Laura DIETSCH - Bastien SALANON

## Tableau récapitulatif des fonctionnalités implémentées

Statut Fonctionnalité	Implémenté intégralement	Implémenté partiellement	Pas du tout implémenté	Commentaires
Affichage de la scène avec des cubes	X			Un pavé de trois cubes de hauteur x nbCubesLignesxnbCubesLi gnes
Caméra	X			Trackball caméra
Curseur	X			N'est pas visible s'il est caché derrière d'autres cubes
Modifier la couleur	X			
Extruder	X			
Creuser	X			
Génération procédurale	X	X		Le poids des points de contrôle n'influence pas la génération des autres cubes (à l'exception du signe)
Modes jour et nuit	X			
Lumière directionnelle	X			
Point de lumière			X	
Édition des lumières			X	
Fonctionnalité additionnelle 1		X		Blocs texturés Début d'implémentation mais pas terminé
Fonctionnalité additionnelle 2	X			Save et Load

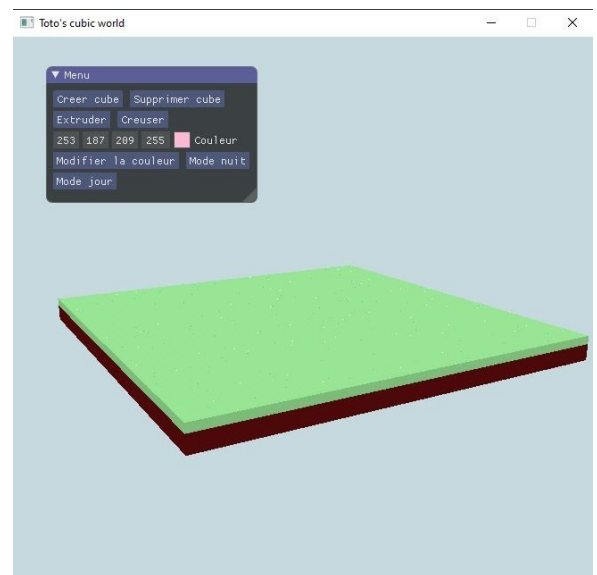
# Architecture des fonctionnalités



## ● Affichage de la scène avec des cubes :

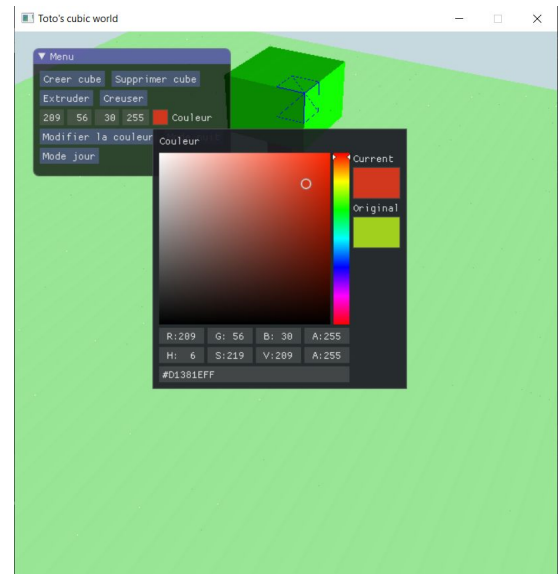
Le fond monde est constitué de trois couches de cubes de couleur unie. Chaque cube du monde est défini par sa position (contenue dans un vecteur qui connaît toutes les positions de tous les cubes existants dans le monde) et par sa couleur (contenue dans un vecteur connaissant toutes les couleurs de tous les cubes existants dans le monde). Lorsqu'on ajoute ou supprime un cube, on met directement à jour le tableau par les méthodes push-back et erase.

La méthode draw de la classe CubesExistants dessine à chaque itération de la boucle de rendu tous les cubes existants en une fois de façon instanciée.



- **Modifier la couleur :**

Lorsqu'on modifie la couleur on utilise la méthode `indiceCube` (de la classe `CubesExistants`) qui permet de savoir quel est l'indice du cube sélectionné avec le curseur. On met à jour directement la couleur du tableau à l'indice récupéré. On utilise un color picker pour sélectionner la couleur.



- **Extruder et creuser**

La méthode `extrusion` comporte un `while` qui incrémente la coordonnée Y du curseur jusqu'à ce que l'espace soit vide. Quand il a trouvé le premier espace vide il crée un cube. De la même manière, la méthode `creuser` va chercher quel est la coordonnée Y du cube le plus haut au-dessus du cube sélectionné et s'arrête dès qu'il rencontre un espace vide, il supprime le cube juste avant cet espace vide.

- **Curseur :**

Le curseur est un cube qui est créé à une position donnée puis détruit de cette position lorsque l'utilisateur bouge le curseur vers une nouvelle position. Le cube-curseur est alors créé à cette nouvelle position (selon la méthode de `CubesExistants`) et ainsi de suite. Le cube-curseur est seulement représenté par ses arêtes de manière à être différencié des autres cubes et rester visible par dessus un cube déjà existant.

- **Trackball caméra :**

Nous avons récupéré la Trackball camera faite dans les TP. La classe `TrackballCamera` contient 3 attributs : la distance entre le centre de la scène et la caméra, l'angle autour de l'axe Y et l'angle autour de l'axe X. La méthode `getViewMatrix` calcule la ViewMatrix en fonction des trois paramètres précédents.

- **Génération procédurale de terrain :**

Donné un nombre fini de points de contrôle et de poids associés on utilise une Radial Basis Function pour générer le terrain dans une zone arbitrairement délimitée. On utilise en particulier une méthode de résolution LU pour le calcul des oméga car c'est la méthode la plus rapide et nous n'avons pas besoin d'une grande précision pour cette fonctionnalité. Pour chaque cube de la zone est calculé son poids en fonction des poids des points de contrôle. Plus un cube est proche d'un point de contrôle de poids  $p$  plus son poids s'approchera de cette valeur  $p$ . On obtient alors un gradient de poids sur l'ensemble de la zone (on peut décider de donner des poids négatifs aux points de contrôle auquel cas certains points de la zone auront également un poids négatif). On décide de dessiner un cube si son poids est supérieur strictement à 0 et de ne pas le dessiner sinon.

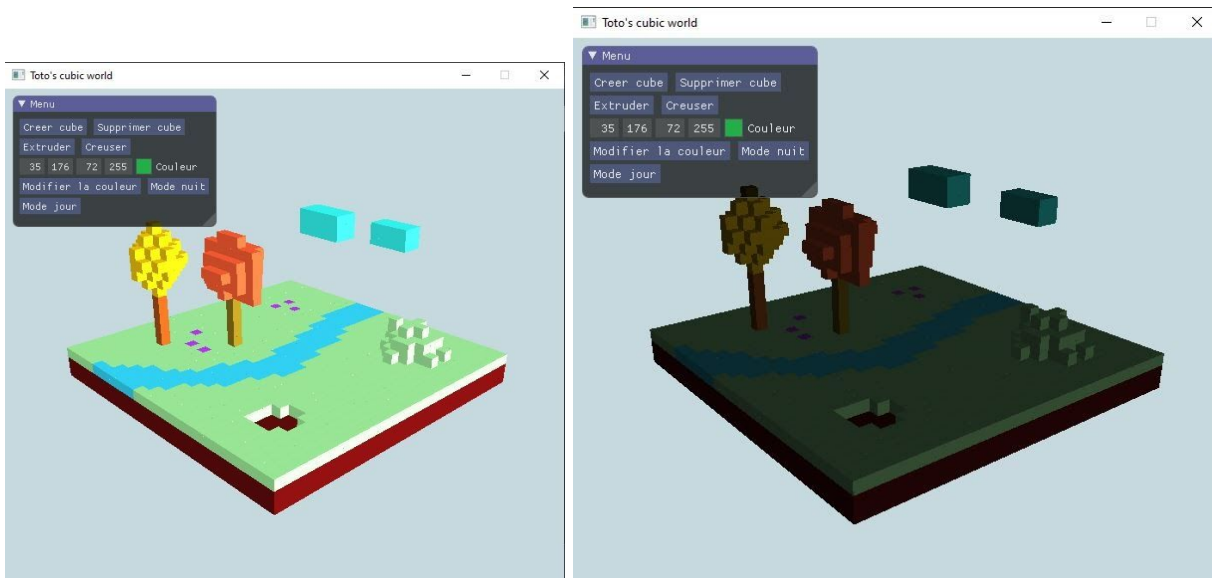
RBF utilisée	Observation
$\exp(-\epsilon \cdot x \cdot x)$ ; // gaussian	génère une sphère centrée autour du point de contrôle dont la taille varie avec epsilon. Plus epsilon est petit; plus le rayon de la sphère augmente.
$\sqrt{1 + \epsilon \cdot d \cdot d}$ ; // multiquadratic	le terrain est rempli de cubes par les PC à poids positif et creusé par ceux à poids négatifs.
$1.0f / (1 + \epsilon \cdot \epsilon \cdot d \cdot d)$ ; // inverse quadratic	génère des blocs ponctuels érodés.
$-\epsilon \cdot d \cdot d$ ;	génère des cuvettes

Pour toutes les illustrations, voir Annexe : Génération procédurale.

Pour les fonctions autres que les gaussiennes il faut jouer avec les points de contrôle (leur position et le signe de leur poids) pour générer le terrain comme souhaité.

- **Ambiance jour/nuit :**

Une lumière (représentée par un vec4 couleur) de faible intensité est multiplié par la couleur de chaque cube du monde existant dans le fragment shader afin de l'assombrir et de créer un effet de nuit. La stratégie inverse est utilisée pour l'effet de jour.



- **Lumière directionnelle :**

Pour chaque fragment de chaque bloc existant dans le monde est défini une normale sortante. On regarde l'angle formé entre cette normale et le segment reliant le bloc lumière au fragment. En particulier, on calcule le produit vectoriel entre ces deux vecteurs : si le produit est grand, cela signifie que le cosinus entre les vecteurs est proche de 1, donc que les deux vecteurs sont proches (au sens de la colinéarité) et donc que le fragment doit être fortement éclairé car directement sous la source de lumière. A contrario, plus l'angle entre ces vecteurs augmente (produit scalaire petit) moins le fragment est éclairé par le bloc lumière. La valeur du produit scalaire obtenu est utilisé pour pondérer l'intensité de l'éclairage du fragment.

- **Fonctionnalité additionnelle : blocs texturés**

Nous avons commencé à suivre un tutoriel (<https://www.youtube.com/watch?v=n4k7ANAFslQ>) pour implémenter des textures sur les blocs grâce à une classe Texture structurée comme sur le diagramme ci-dessous. Il aurait fallu modifier plusieurs attributs de la classe CubesExistants pour inclure les coordonnées de textures et adapté le draw call en conséquence. Nous n'avons pas eu le temps de finir d'implémenter cette fonctionnalité donc elle n'est pas incluse dans le projet.

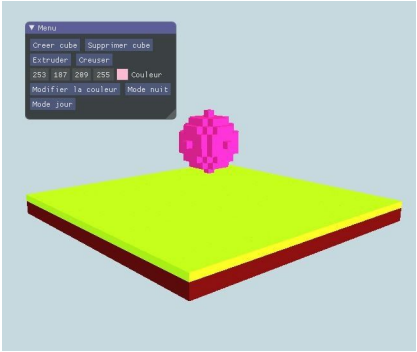
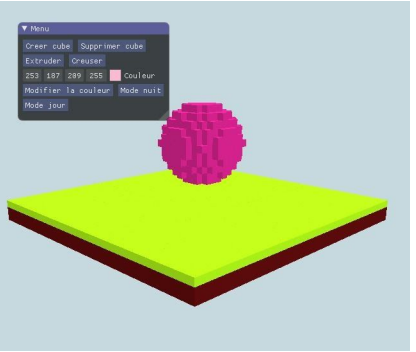
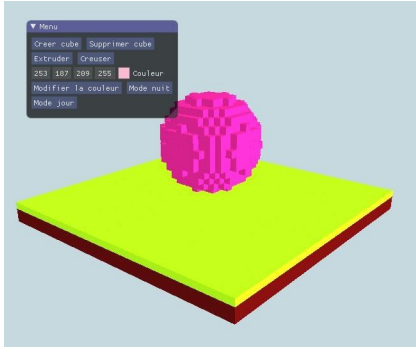
Texture
-m_RendererID : unsigned int
-m_FilePath : std::string
-m_LocalBuffer : unsigned char*
-m_BPP : int
-m_Height : int
-m_Width : int
+Bind(unsigned int slot=0) const : void
+Unbind()

- **Sauvegarde**

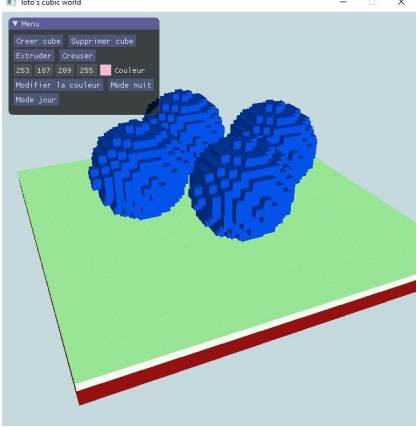
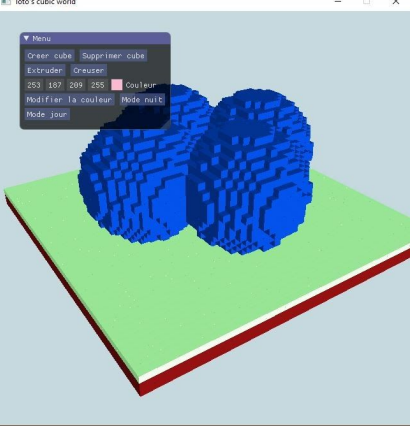
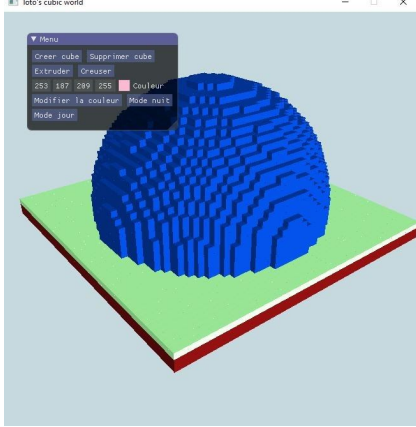
Dans un fichier texte, on stock dans un fichier texte les coordonnées de tous les cubes existants et leur couleur. Chaque ligne du fichier texte correspond à un cube existant.

Pour load, on lit le fichier texte ligne par ligne et on donne à CubesExistants toutes les infos dont il a besoin.

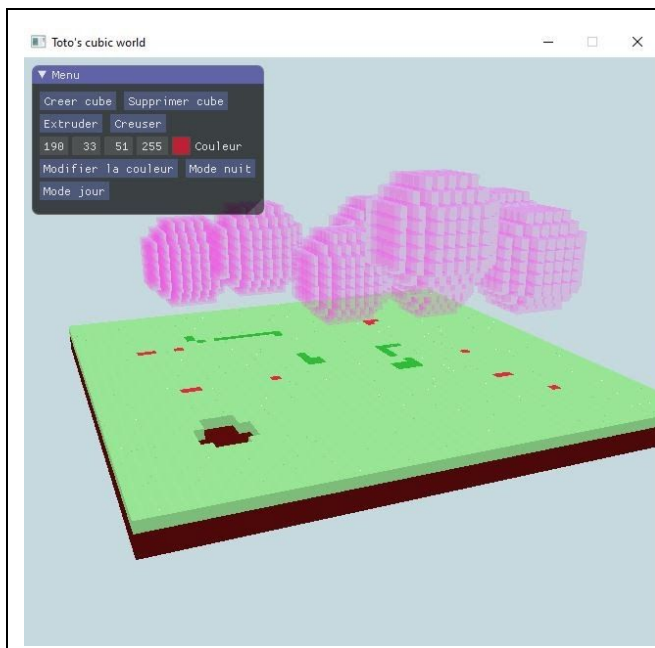
Annexe : Génération procédurale de terrain

		
exp(-0.4*d*d)	exp(-0.1*d*d)	exp(-0.05*d*d)

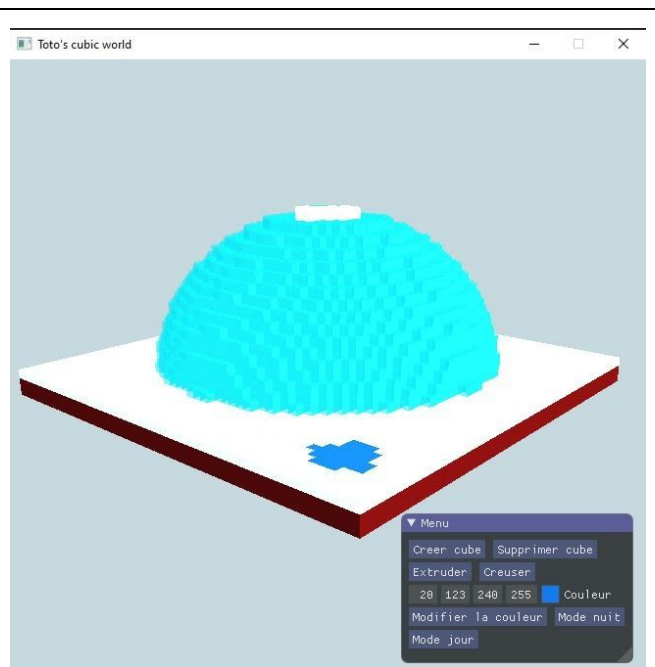
  

		
exp(-20*d*d) et 4 PC	exp(-10*d*d) et 4 PC	exp(-2*d*d)

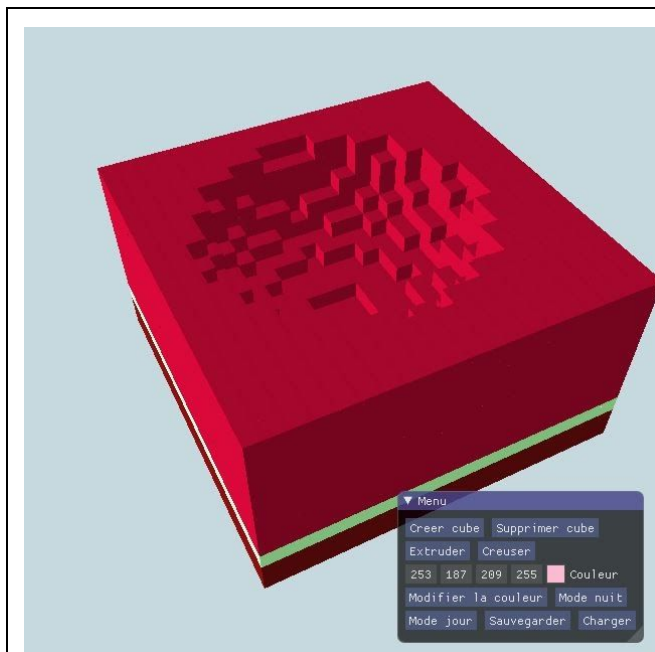




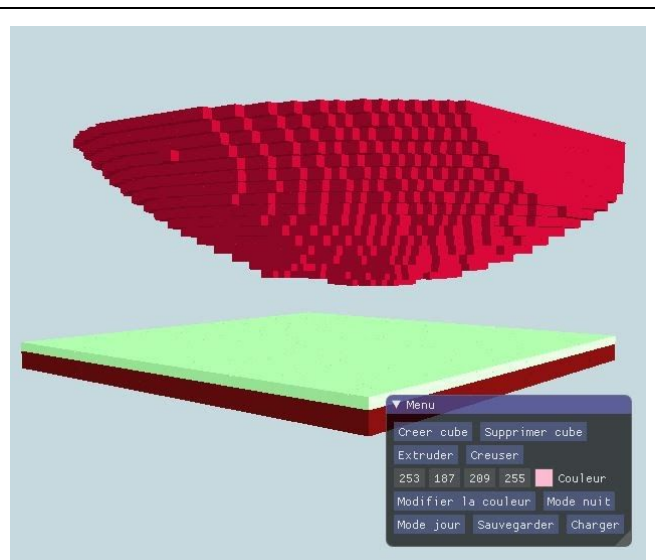
Nuages par RBF ( $\exp(-30 \cdot d \cdot d)$ ) et 8 PC



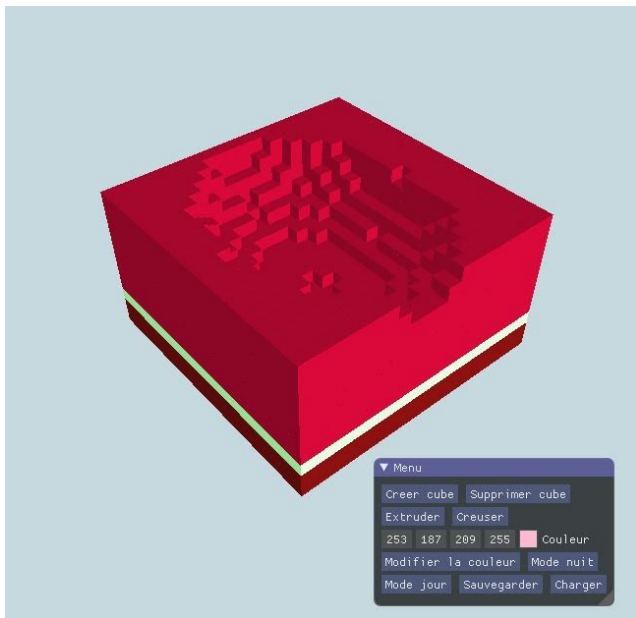
Igloo par RBF ( $\exp(-2 \cdot d \cdot d)$ )



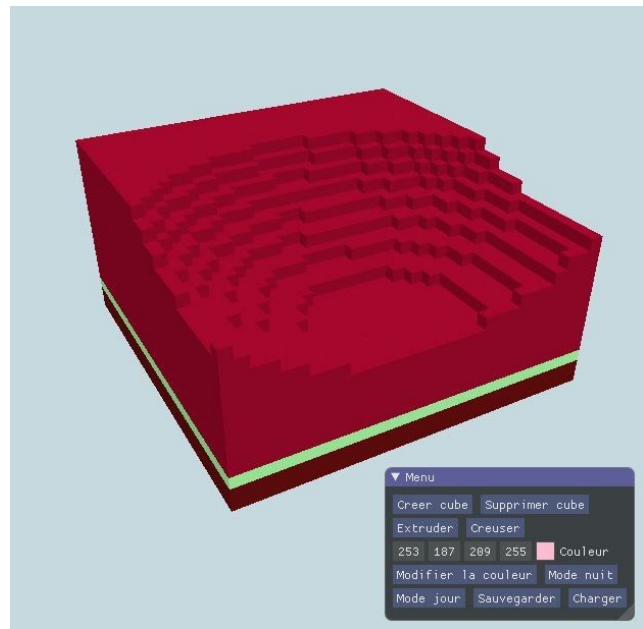
$\sqrt{1 + 0.2 \times 0.2 \times d \cdot d}$  3 PC 1 neg.



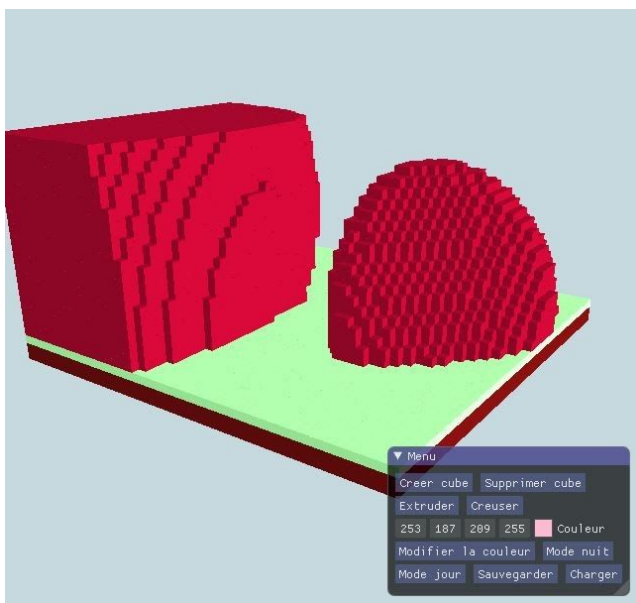
$\sqrt{1 + 0.2 \times 0.2 \times d \cdot d}$  5 PC 3 neg.



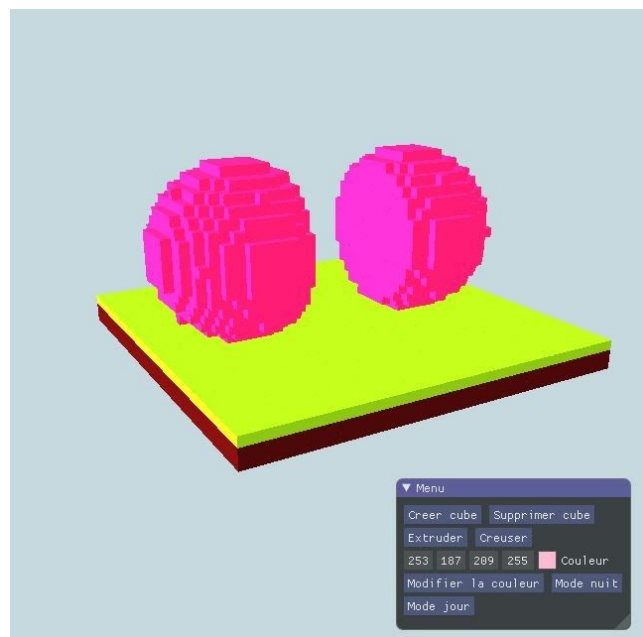
-0,4xdxd avec 3 PC dont 1 neg



-0,01xdxd avec 5 PC dont 3 neg



$1.0f \div (1 + 0.2 \times 0.2 \times d \times d)$  avec 5 PC dont 3 neg



gaussienne; 3 PC dont 1 a poids négatif

## Difficultés rencontrées

Nous n'avons pas pu implémenter tout ce que nous aurions voulu dans la mesure où chaque nouveau problème nous demande un temps conséquent pour être surmonté étant donné notre manque d'aisance à tous les deux en programmation. Nous avons eu du mal à faire les TP et n'étions pas très avancés ce qui nous a fait défaut pendant l'implémentation de ce projet.

## Parties individuelles

### **Bastien :**

J'ai trouvé le projet ambitieux, instructif et complet. J'aurais aimé faire plus mais tout ce que j'ai pu entreprendre m'a pris un temps disproportionné, ce qui a été particulièrement frustrant pour moi. Réfléchir sur ce projet m'a tout de fois beaucoup appris étant donné que je partais d'un niveau quasi inexistant en programmation en septembre. Je ne pensais honnêtement pas que nous arriverions à aller aussi (relativement) loin dans le projet.

### **Laura :**

J'avais beaucoup d'appréhension quant à la réalisation de ce projet car mon binôme et moi n'étions pas très avancés dans les TP, pourtant nous n'avons loupé aucun cours. Le fait de travailler toute une semaine à l'université nous a permis de nous entraider avec les autres groupes et de progresser. Je trouve que le sujet du projet est bien pour évaluer nos compétences en OpenGL car assez complet et j'ai apprécié qu'ils soient regroupés avec la programmation et les maths info pour avoir moins de projets différents. En revanche je le trouve quand même trop dense et il me paraît très difficile de le réaliser en répondant à toutes les consignes. Ce n'est pas vraiment le délai qui me paraît problématique mais plutôt que nous n'avons pas assez de temps libre pour le réaliser en entier compte tenu des examens et des autres projets conséquents que nous avons à l'IMAC.

Je suis très contente du résultat car, au vu de notre niveau en synthèse d'image il y a quelques semaines, nous ne nous sommes pas démontés et avons réussi à implémenter la majeure partie du projet et sans bug. J'ai pu beaucoup progresser en programmation également et je me sens plus à l'aise pour implémenter des classes.