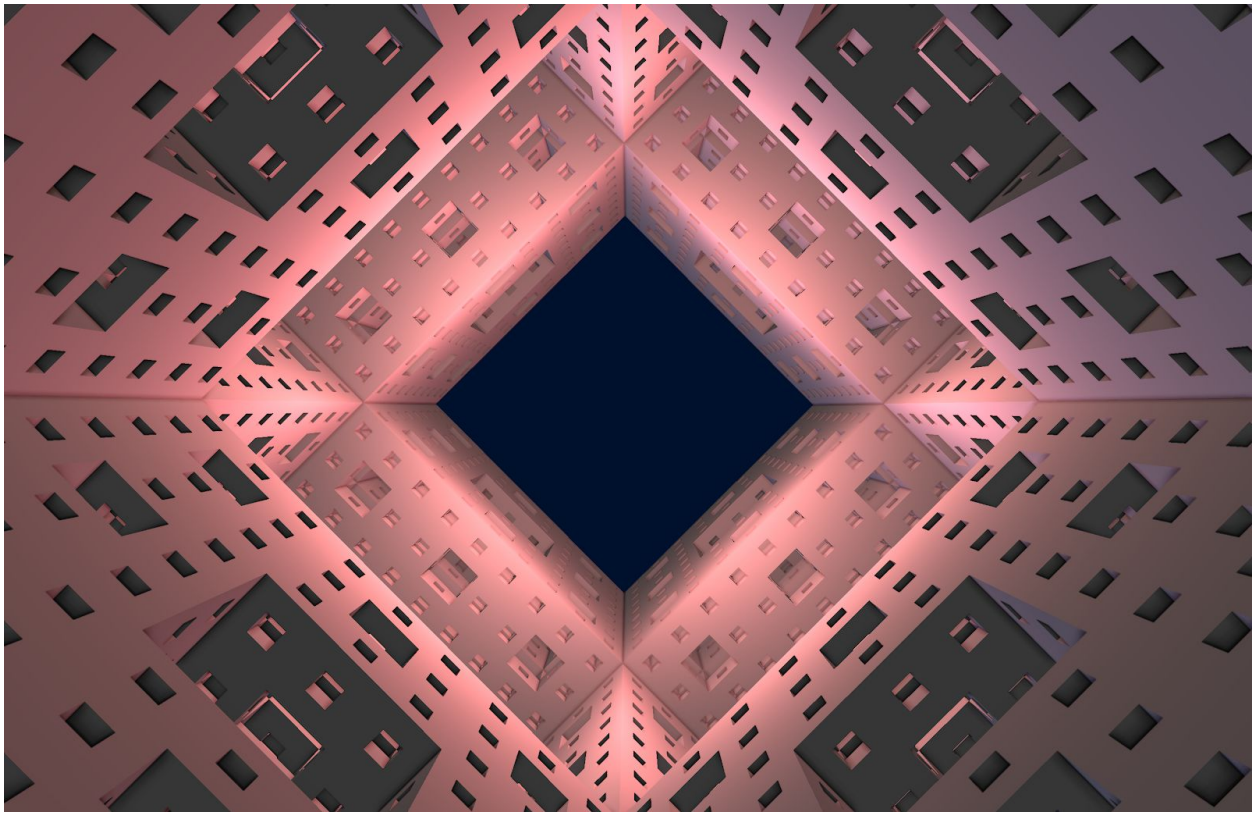
C++ - Synthèse d'images - Mathématiques
World IMaker

Jules Fouchy, Thomas Geslin, Victor Julien



Liste des fonctionnalités	Implémentation	Précisions sur les éléments implémentés
Afficher les cubes	Complète	-Rendu instancié
Système de matériaux	Complète	- Les shaders ont des paramètres (couleur etc.) qui sont parsés automatiquement et pour lesquels on crée des sliders - Ajout et modification de shaders/liste de paramètres pour ce shader - Modifier un matériau modifie tous les cubes concernés
Ajouter/supprimer des cubes	Complète	- Grille 3D pour retrouver les cubes en O(1) - Buffers pour envoyer à OpenGL
Caméra	Complète	- Zoom avant/arrière - Translation - Rotation - Recentrage sur le curseur -Utilise le pattern State pour switcher entre les modes “repos”, “translation” et “rotation”
Positionner le curseur à la souris	Complète	-Ray casting
Lumières ponctuelles	Complète	- Composantes diffuse

et directionnelles		et spéculaire (modèle de Blinn-Phong) -Autant de lumières qu'on veut
Occlusion ambiante	Complète	-Screen Space Ambient Occlusion
Ombres	Partielle	-Shadow mapping -(Seules les lumières directionnelles font des ombres)
Export d'images	Complète	-On peut choisir la taille de l'image (le ratio de la vue est préservé automatiquement) et le chemin de fichier -Lance un warning si on essaye d'écrire sur un fichier déjà existant - (L'explorateur de fichier ne marche que sous windows)
Outils	Complète	-Plusieurs forme de brush (cube, sphère, menger sponge (avec éventuellement des scales différents selon les axes)) qui permettent d'ajouter ou supprimer des cubes, ou peindre les cubes existants - Outils RBF avec options, sélection des points de contrôle à la

		souris
RBF	Complète	- Choix de la fonction qui module la distance, avec des paramètres - Template pour pouvoir choisir des vec2 ou des vec3 comme points de contrôle - Deux interprétations de la RBF : height map (points de contrôle en 2D et la valeur de la RBF est la hauteur en ce point), et distance field (points de contrôle en 3D et la valeur de la RBF est la distance à la scène (donc on place des cubes quand elle est négative)) - Possibilité de placer des cubes seulement à la surface des formes décrites par la RBF, ou alors dans tout le volume
Historique	Complète	-Plusieurs historiques en parallèle : pour les cubes, les matériaux et les lumières.
Menus ImGui	Complète	GUI pour toutes les fonctions sus-mentionnées, avec plein de paramètres pour chacune

Afficher les cubes :

Nous utilisons du rendu instancié : on envoie une seule fois les vertices qui décrivent un cube (stockés de manière statique), puis on envoie un buffer contenant les positions de tous les cubes.

Système de matériaux:

L'objectif était de donner une grande liberté à l'utilisateur quant à l'apparence des cubes (d'autant plus grande qu'il a l'habitude des maths, il faut bien l'avouer). L'idée est donc de permettre à l'utilisateur d'écrire ses propres *fragment shaders* pour contrôler la couleur (albedo) de chaque pixel du cube. Il peut définir des paramètres dont les valeurs seront ensuite accessibles et modifiables via l'interface du logiciel, et a aussi accès à plein d'informations sur le fragment : position dans le monde, position dans le cube, position dans la face (aka Texture Coordinates), et ID de la face (left, right, bot, top, back ou front).

En pratique, et toujours dans l'optique de limiter le nombre de DrawCall, on a une classe qui regroupe un Shader ainsi que tous les Materials qui lui sont associés ; elle stocke aussi un CubesGroup qui gère le buffer instancié des positions, ainsi qu'un buffer instancié de MaterialIndex. Ensuite on envoie un uniform array de *Struct De Paramètres Définis Par l'Utilisateur* (chaque Material est chargé de remplir une case de cet array), et le MaterialIndex sert à chaque cube à savoir laquelle de ces structs est celle qu'on lui a associé.
On a donc un DrawCall par shader, et on peut difficilement faire mieux.

Gestion des shaders et des uniforms

-ShaderLibrary

Tous nos shaders sont stockés dans un *std::vector* (la ShaderLibrary),
Ceci essentiellement pour pouvoir stocker des “références” vers les Shaders grâce à un index, plutôt qu’avec une vraie *Shader&* qui sont toujours un peu pénibles à stocker comme membre d’une classe.
Cette ShaderLibrary gère donc le chargement, rechargement (important pour que l’utilisateur puisse modifier son Shader et le recharger pendant que l’application tourne !) et l’envoi des uniforms, qui se fait via des UniformUpdateList.

-UniformUpdateList

Vu le nombre important de shaders que nous utilisons, il fallait un moyen simple de communiquer entre, par exemple, la caméra qui doit envoyer des uniforms, et tous les shaders qui ont besoin de ces infos (ceux définis par l’utilisateur, plus le shader pour le curseur).
Chaque Shader (représenté par son ID dans la Library) peut donc s’abonner à une (ou plusieurs) UniformUpdateList ; et les classes qui ont des uniforms à envoyer les transmettent à l’UpdateList correspondante, qui redispatche ensuite à tous les shaders abonnés (et elle stocke aussi les Uniforms, au cas où un abonné arriverait en cours de route, ou quand un Shader est rechargé, il faut pouvoir leur envoyer toutes les Uniforms d’un coup)

Ajouter/supprimer des cubes:

Nos informations sur les cubes sont stockées en deux endroits : une grille 3D (la CubesMap) qui indique pour chaque position possible s’il y a un cube ou pas (plus précisément elle indique dans quel CubesGroup est

rangé le cube, s'il existe), et des buffers (les CubesGroup) qui contiennent des positions de cubes existants, en vrac mais contigus en mémoire, ce qui permet de les envoyer facilement à OpenGL. Il y un CubesGroup par Shader comme précisé précédemment, qui contient tous les cubes dont le matériau correspond à ce Shader.

- Ajout

Tout commence par la CubesMap ; on regarde si la position où on veut ajouter est cube est déjà occupée : si c'est le cas il faut d'abord aller dans le CubesGroup correspondant et supprimer le cube. Ensuite on peut écrire l'emplacement (shader+matériau) du nouveau cube dans la CubesMap, et aller l'ajouter dans le bon CubesGroup.
Pour ajouter dans un CubesGroup, il suffit de push_back la position dans le *std::vector*, puis de renvoyer le buffer actualisé à OpenGL.

- Suppression

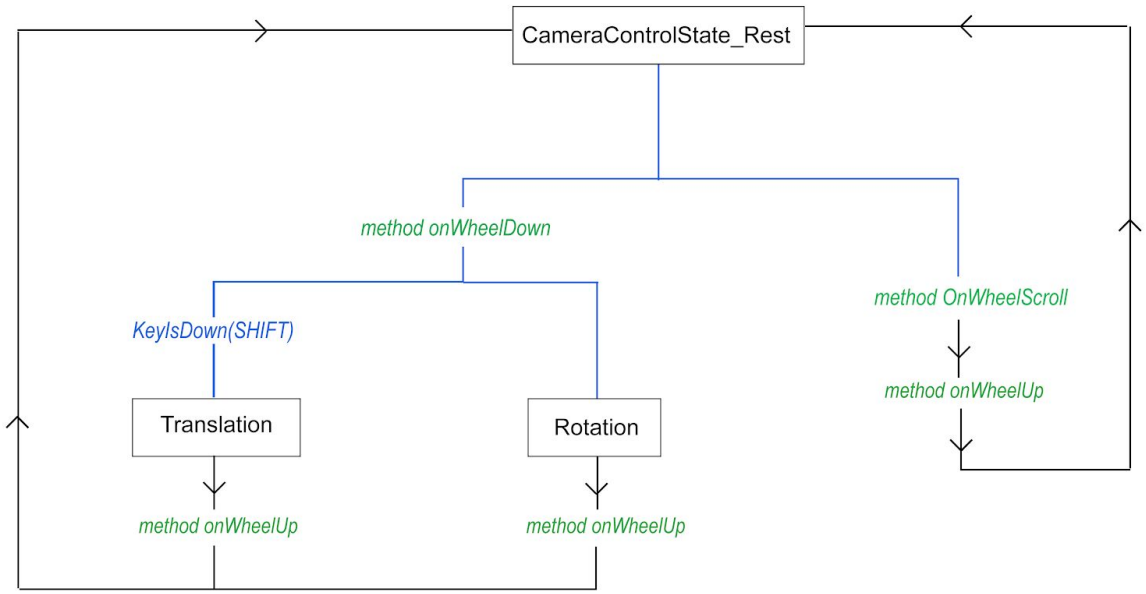
De même, on commence par la CubesMap : si le cube n'existait pas il n'y a rien à faire, et sinon il faut aller dans le bon CubesGroup pour supprimer le cube.
Pour supprimer dans un CubesGroup, il suffit de retirer l'élément correspondant dans notre buffer (qui est un std::vector : on fait donc un swap avec le dernier élément, puis un pop_back). Ceci dit retrouver notre cube dans le buffer est embêtant, puisque les données y sont en vrac. On utilise donc une *std::unordered_map* qui associe un cube (représenté par sa position, un *glm::ivec3*) à son indice dans le buffer. Il suffit de bien s'assurer que cette map est toujours à jour, et le tour est joué !
(et bien sûr il ne faut pas oublier de renvoyer le buffer mis à jour au GPU)

Caméra:

La gestion de la caméra est organisée selon un design pattern de type "state" et géré par un ensemble de fichiers:

- CameraControlState_Rest

- CameraControlState_Translation
- CameraControlState_Rotation



Architecture des fonctionnalités de la caméra

Positionner le curseur à la souris:

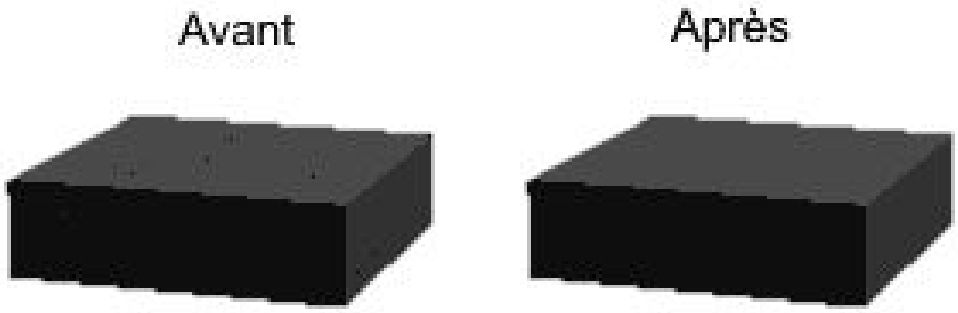
On utilise du ray casting, adapté à notre cas : on n'a que des cubes, régulièrement espacés sur une grille.

On a donc une position, qui commence à la position de la caméra. On teste si la case3D de la grille contenant cette position contient un cube ou pas : si oui, on peut s'arrêter. Sinon, on cherche la prochaine case 3D par lequel passe notre rayon (rayon entre la caméra et le curseur de la souris) : pour cela il suffit de regarder quelle face de la case3D à l'intérieur de laquelle on est est intersectée par le rayon. On se place ensuite dans cette nouvelle case (quelque part le long du rayon toujours) et on réitère :

on regarde si cette case contient un cube, et sinon on avance vers la prochaine case.
On s'arrête quand on trouve un cube, ou qu'on atteint la limite du monde.

Lumières

Pour éviter qu'avoir trop de lumières ne cause du lag, nous avons été poussés à passer en rendu différé :
On fait une première passe pour savoir où est la géométrie, et on stocke les informations nécessaires aux calculs de lumière dans des textures (albedo, normales, et propriétés lumineuses du matériau (brillance etc.))
Puis une deuxième passe s'occupe des calculs de lumière coûteux.
Le rendu différé nous a de plus été utile plus tard pour l'ambient occlusion, et nous a permis de corriger des défauts de rasterization dans la NormalMap. Nous ne savons pas vraiment à quoi ils étaient dus, mais pour les corriger nous utilisons un shader appliqué à la texture NormalMap qui regarde les 8 pixels autour d'un pixel donné, et choisis comme normale celle qui est le plus présente autour.



Ensuite, pour pouvoir envoyer un uniform array de taille variable à OpenGL, il nous a fallu mettre en place une recompilation du shader à chaque fois qu'on ajoute une lumière, et avant de l'envoyer à la compilation on lis le fichier et remplace un tag du genre #nbPointLights par le nombre

de lumières actuel. De sorte qu'OpenGL voit toujours des array de taille bien connue et ne râle pas.

Occlusion ambiante et Ombres :

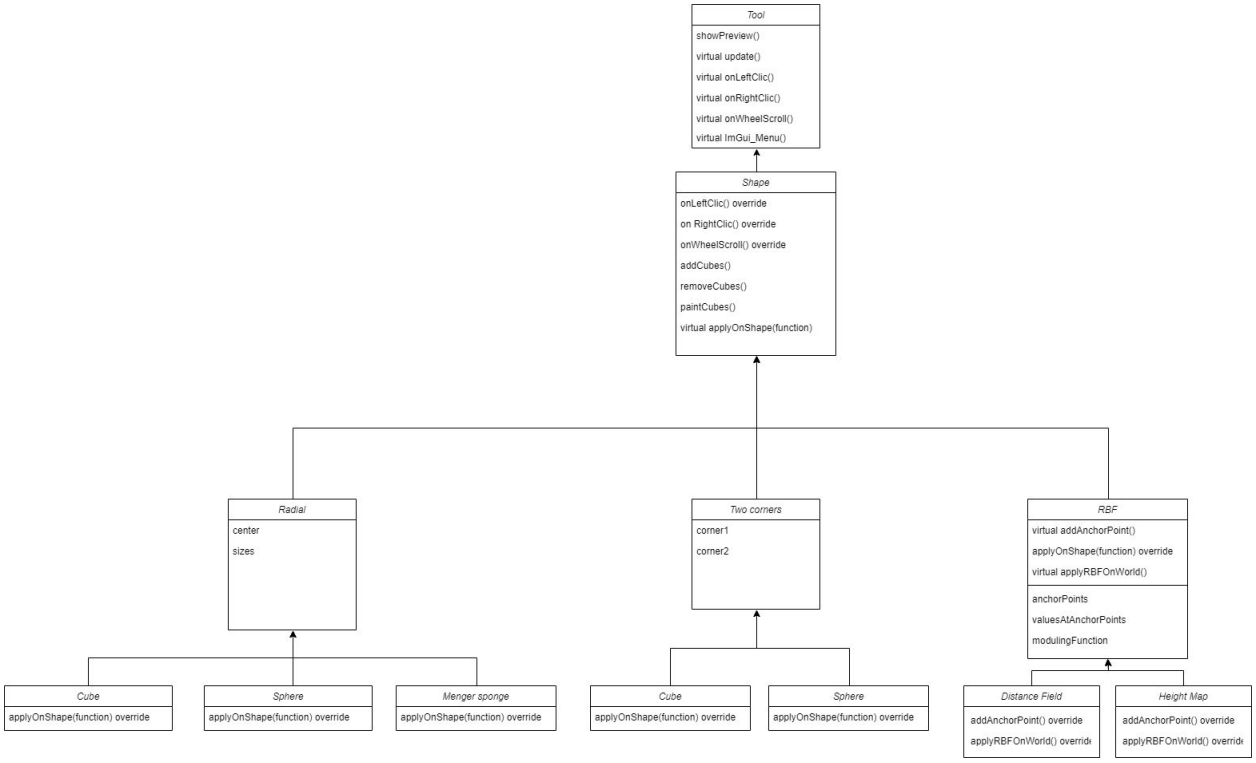
Que dire sinon que nous avons suivi ces tutoriels à la lettre et qu'ils sont géniaux ?
https://learnopengl.com/Advanced-Lighting/SSAO
https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

Export d'images

On utilise un frameBuffer pour pouvoir créer une image de la taille voulue.

Outils

Beaucoup d'héritage, puisque les outils partagent beaucoup de comportements : appliquer une action sur un ensemble de cube (sphère, pavé…), que ce soit ajouter, supprimer des cubes, ou changer le matériau. Les classes tout au bout de la hiérarchie se contentent d'implémenter la manière donc les cases du monde sont parcourues, et le test qui détermine si on applique l'action (reçue comme pointeur de fonction).



Radial Basis Functions:

En réutilisant les TPs vus en C++, nous permettons à l'utilisateur de choisir entre différents types de Radial Basis Functions ("RBF") ainsi que leur paramètres à l'aide d'un menu ImGui,

- Gaussienne $e^{-|\lambda x|^{\alpha}}$
- Inverse $\frac{1}{1+|\lambda x|^{\alpha}}$
- Identité
- Quasi Identité (aka Multi Quadratique, généralisée) $(1+|\lambda x|^{\alpha})^{\frac{1}{\alpha}}$

Ces fonctions sont regroupées dans le fichier "Rbf". Une fois le type de RBF sélectionnée, le bouton de confirmation de la fenêtre lance la méthode OnConfirmation, héritée et redéfinie par méthode virtuelle de la classe "PopUpWindow". La partie calculatoire de ces fonctions appliquées

à l'ensemble des points de contrôles (spécifiés par l'utilisateur) est gérée par le fichier "Tool_RBF".

Historique

Le constituant de base de l'historique est l'Action : deux pointeurs de fonction qui expliquent comme faire et défaire une action.

Ensuite, l'historique stocke des actions, mais regroupées entre elles pour former des "undo group" : si par exemple on génère du terrain procéduralement, on aura plein d'actions "addCube", mais on veut pouvoir annuler tout le terrain en un seul CTRL+Z, d'où la nécessité des undo groups.

Pour sauvegarder quelque chose dans l'historique, on appelle donc "beginUndoGroup()", fais ce qu'on a à faire, en ajoutant aussi les actions correspondantes dans l'historique, puis "endUndoGroup".

A noter aussi qu'on dispose de plusieurs historiques en parallèle : pour les cubes, les matériaux et les lumières, ce qui évite des conflits du genre "en fait j'aimais bien la couleur du tout début, mais si j'y reviens tous les cubes que j'ai ajouté entre temps vont être annulés". (D'ailleurs je me demande pourquoi si peu de logiciels proposent un tel système d'historiques en parallèle, alors que c'est tellement plus pratique et ne rajoute aucune complexité au code)

L'historique courant (sur lequel s'applique le CTRL+Z/Y) est le dernier à avoir reçu un changement, mais on peut aussi si besoin le changer manuellement en allant dans le menu "Histories".

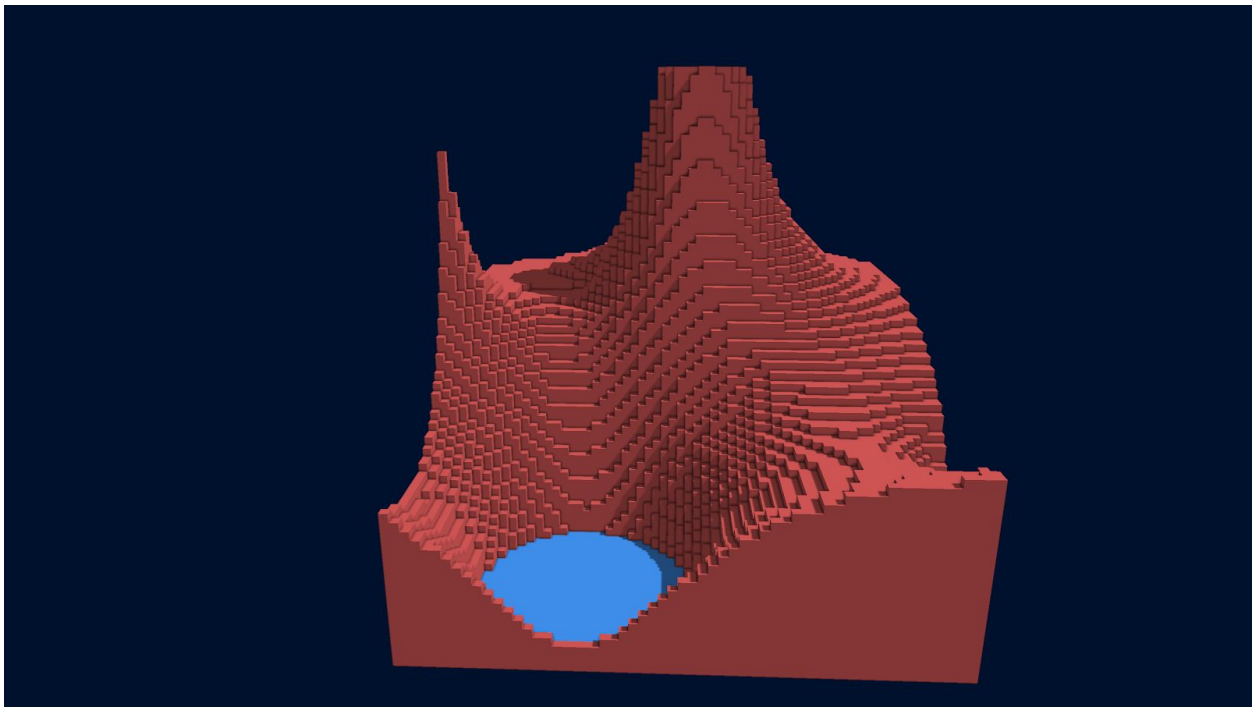
Menus ImGui

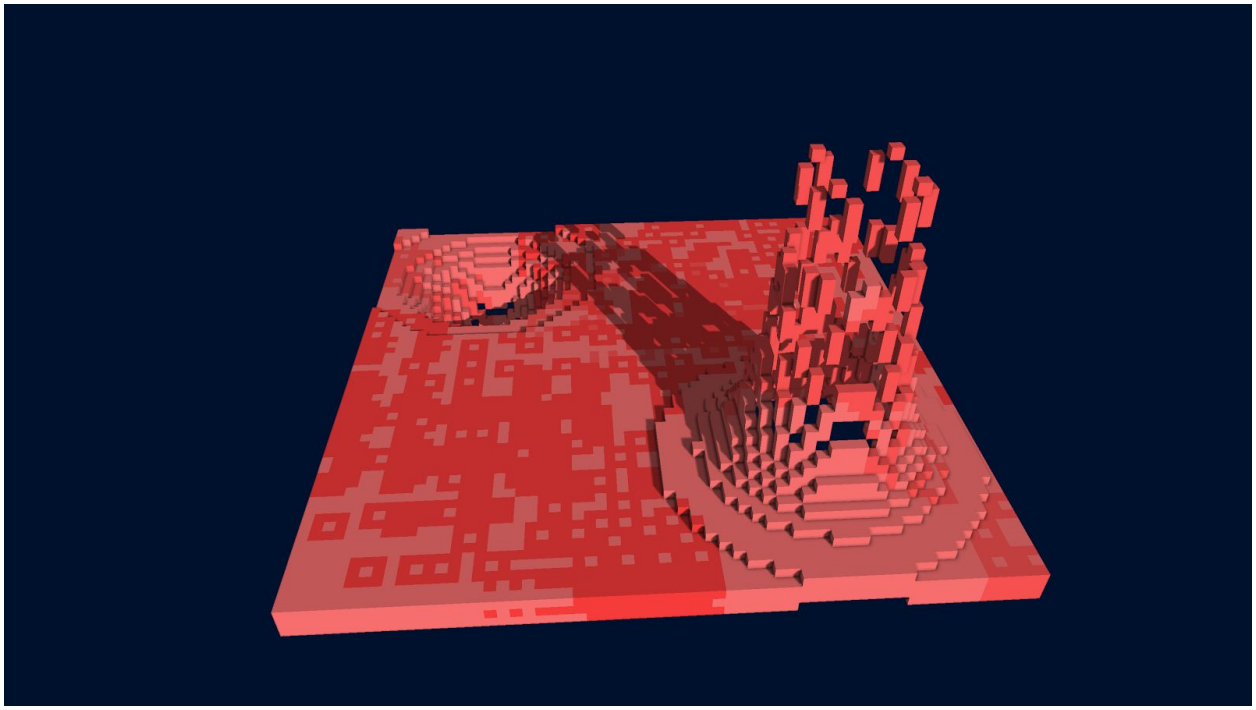
Chaque classe est responsable de créer les outils ImGui dont elle a besoin (slider, checkbox, color picker …) puisqu'elle dispose des pointeurs

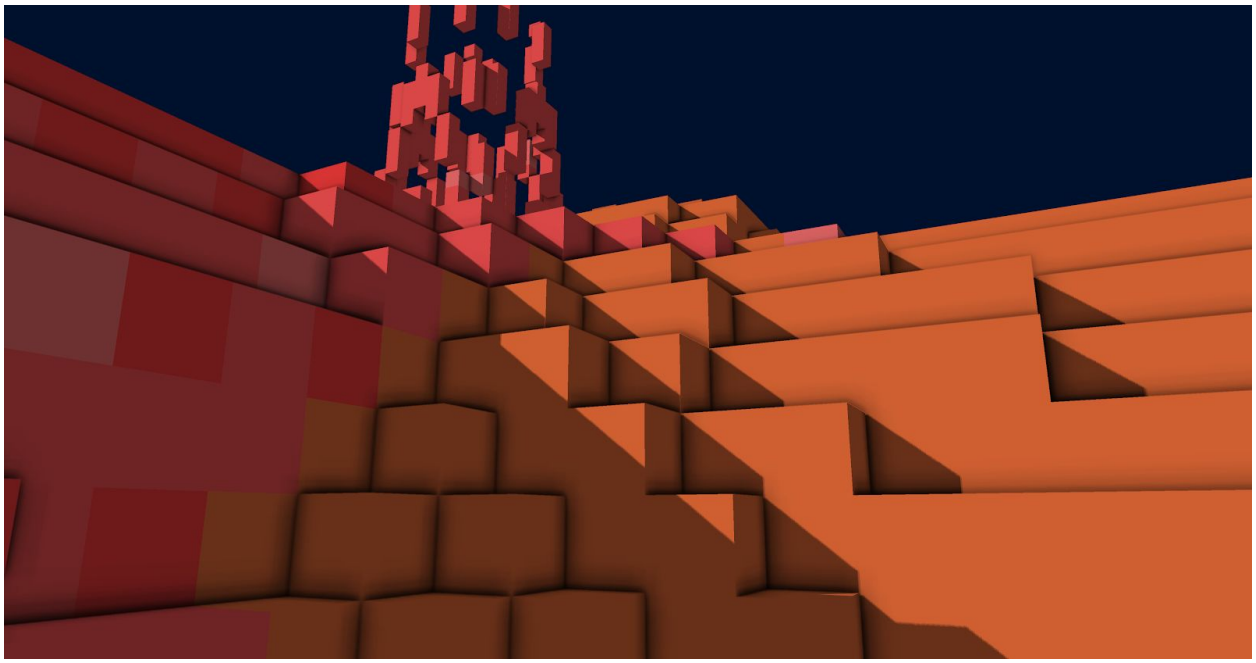
sur ses membres. Par contre, elles ne gèrent pas la création des fenêtres/menus, qui est géré par le “main” (App) : le main s’occupe de la disposition, puis demande à chaque classe de remplir le contenu.

Scènes générées avec les RBF

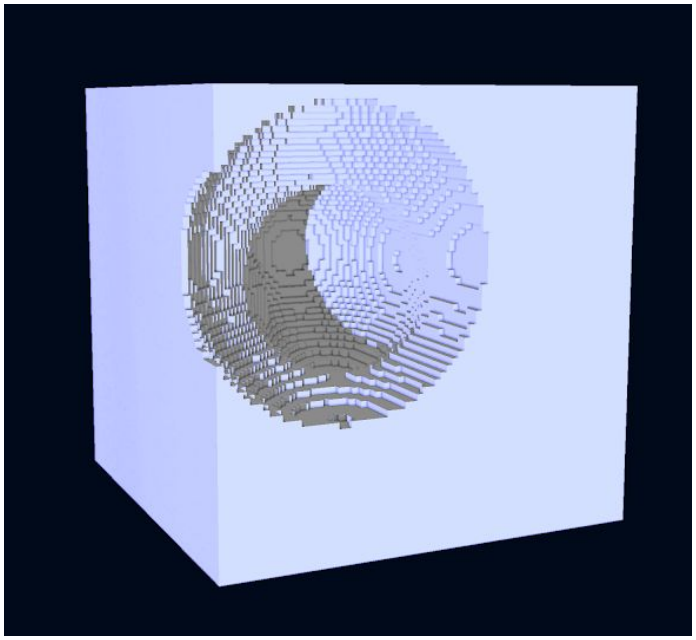
Height map :

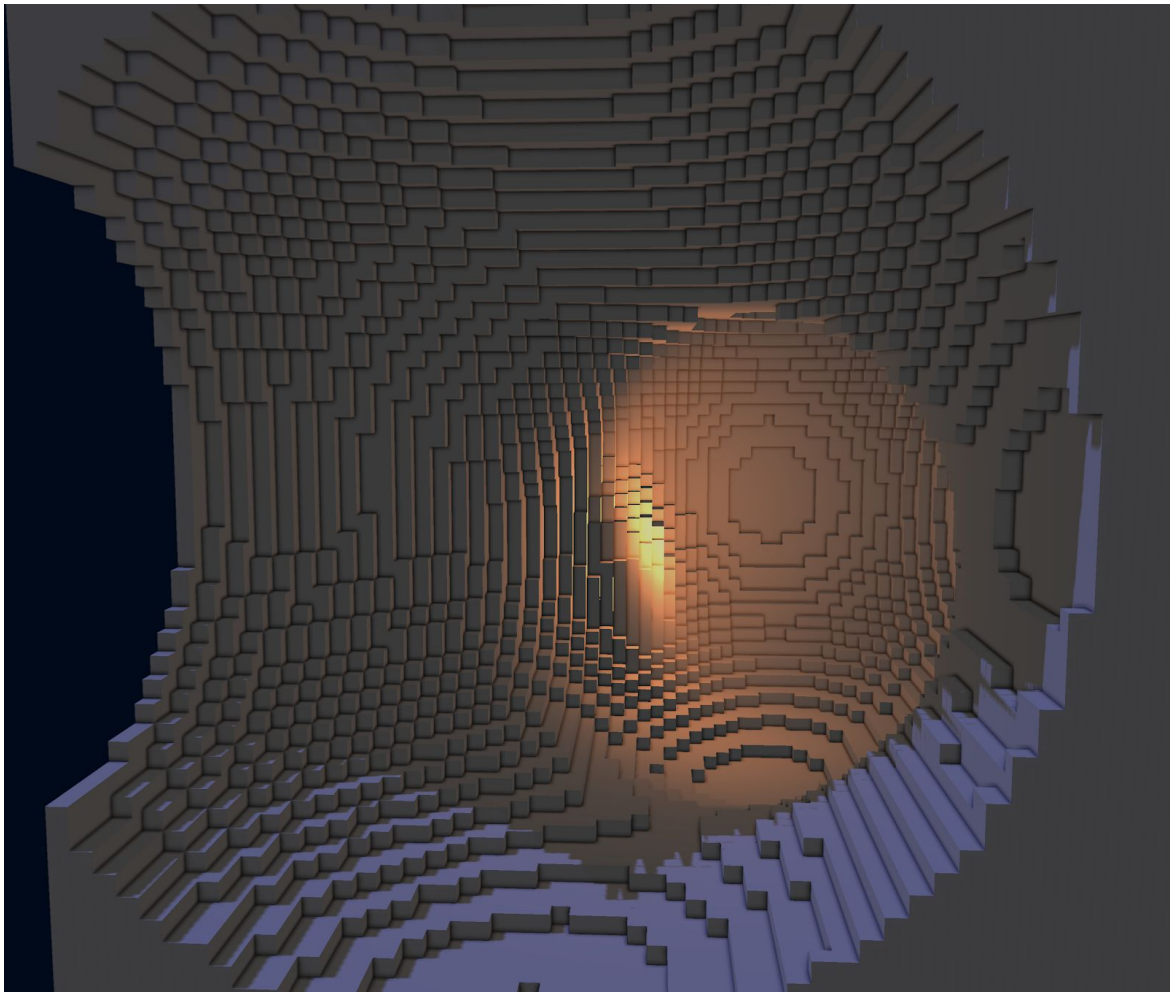


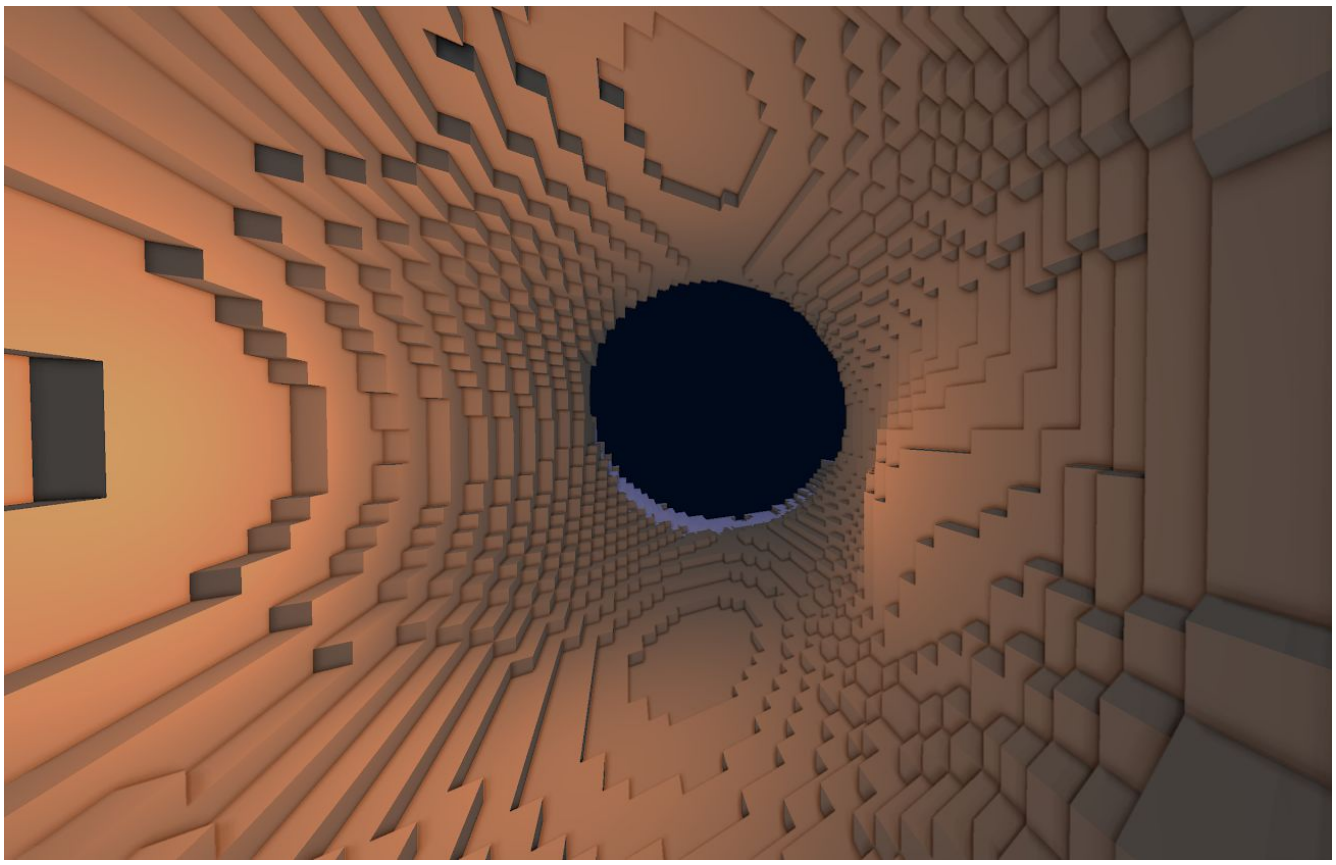




Distance field :







De par sa nature radiale, la RBD 3D (distance field) permet essentiellement de générer des formes/creux ronds, tandis que la height map est plus adaptée pour générer du relief (terrain, montagne…)

La vitesse de décroissance extrême de la gaussienne la rend difficile à contrôler, et l'influence des points est souvent tellement limitée à leur voisinage proche qu'on n'obtient que des sphères centrées en les points d'ancrage. La fonction inverse s'est révélée elle beaucoup plus maniable et intéressante. (éventuellement la gaussienne est utilisable en mode Height Map pour générer des pics très pointus, mais encore une fois l'Inverse est généralement plus esthétique.)

Quant aux fonctions croissantes (Identité et Quasi Identité), nous avons trouvé plus difficile de comprendre et prédire leur comportement,

rendant leur utilisation un peu aléatoire ; de plus, les formes générées ne semblent pas fondamentalement différente de ce que peut obtenir avec les gaussienne/inverse.

Difficultés rencontrées

Il manquait du temps pour ajouter un langage de scripting snif T_T
Honnêtement, nommer ses classes c'est pas simple, il faut trouver un nom à la fois descriptif, compréhensible mais court.

Parties Individuelles :

Jules :

J'ai utilisé des templates pour la première fois, fait et appris à ne plus faire les erreurs de base qui cassent tout (avec des messages d'erreur pas toujours clairs) et savouré leur côté pratique. De même, je me suis efforcé à utiliser plus de fonctionnalités du C++ moderne (les *unique_ptr* notamment que j'ai utilisé pour la première fois ; et implémenter un itérateur). Découvrir les *move constructor* a aussi été une grande révélation pour moi (surtout après avoir eu des problèmes avec des *vector* qui appelaient le *default copy constructor,* donc détruisait mes objets OpenGL et pointeurs).

Ça a aussi été un plaisir d'implémenter le pattern State, que j'ai découvert en cours d'Architecture Logiciel et qui rend le code très agréable (surtout après avoir fait dans un projet précédent de la gestion d'état avec des booléens dans tous les sens et tout dans un même fichier (et un peu plus que trois états !), donnant un code immonde et pas vraiment robuste (disons qu'il ne fallait pas trop spammer les touches si on voulait éviter la *segfault*)).

Ce projet m'a permis de faire mieux ce que j'avais déjà fait dans d'autres projets, et d'améliorer des classes que j'utilisais déjà et qui me

resserviront (Shader, ShaderLibrary, Uniform, UniformUpdateList, Historique, PopupWindow, File...)

C'était aussi la première fois que je faisais de la 3D en Rasterization, et je suis content d'avoir pu m'essayer à un certain nombre de techniques (rendu instancié, rendu différé, screen space ambient occlusion, shadow maps).

J'ai aussi appris à utiliser CMake ! (et à l'utiliser proprement avec VisualStudio ^^) J'ai passé une bonne journée à écrire le CMake de mes rêves, et j'en suis content (à 99%). Dans la même veine, j'ai utilisé plus souvent les branches sur Git, et ça m'a bien servi !

Même s'il y a forcément des regrets à la fin d'un projet quand on pense à tout ce qu'on n'a pas eu le temps d'implémenter, je suis très satisfait de notre petit logiciel qui m'aura permis de produire quelques belles images dont je suis fier.

Thomas :

J'ai aimé la mise en pratique de nos connaissances dans la réalisation d'un logiciel avec rendus visuels, et la réflexion liée à l'optimisation du code. Un logiciel comme celui-ci étant interactif et en temps réel, il était important de se placer du point de vue de l'utilisateur pour rendre son utilisation fluide. J'ai également apprécié les méthodes et idées proposées par Jules pour améliorer/corriger mon code qui fonctionnait mais ne s'avérait pas optimale, notamment avec l'héritage des classes de Tools et des fenêtres PopUp ImGui, qui étaient vraiment redondantes. Les outils utilisés pour le workflow de ce projet ont été cruciaux et nous a probablement éviter de très nombreux problèmes de versioning: Git, Gitkraken, Conan, CMake) La fonctionnalité de gestion des matériaux est celle qui m'a le plus satisfait une fois correctement implémentée. L'initialisation du projet était ce qui me semblait le plus compliqué, mais l'utilisation de certains travaux précédents de Jules nous a permis de nous plonger assez rapidement et facilement dans le vif du sujet, encore merci à lui!

Victor :

Lors du précédent projet d'informatique, je n'avais malheureusement pas trouvé de binôme pour m'accompagner. Alors quelle joie de se retrouver dans une telle équipe pour attaquer sérieusement le S3 de l'IMAC.

Pour entamer le projet dans les meilleures conditions, nous avons décidé de travailler sur le même environnement : Virtual Studio. Je ne connaissais pas bien cet environnement de développement auparavant puisque je travaillais sur Linux - via une machine virtuelle qui plus est.

Après l'installation des outils nécessaires, des problèmes de versioning, de chargement de symboles et de CMake nous ont un temps ralenti au commencement du projet, mais nous avons résolu ces soucis de mise en route à force de reconfigurer le projet et en passant par un fichier solution .sln , ce que je n'avais jamais fait jusque là.

Bien que je n'étais plus un novice dans l'utilisation de Git, Jules m'a fait découvrir GitKraken, et j'ai vite compris son utilité dans un projet en groupe.

Pour la partie programmation, mes camarades m'auront aidé dans bons nombres des problèmes que j'ai pu rencontrer, notamment lors du développement de la fonction addCubes; mais les galères furent vite oubliées lorsque l'on réussit - dans les premières versions de l'app - à placer les cubes et à se déplacer avec les touches du clavier.

Finalement, je retiens beaucoup de positif de ce projet, bien que la programmation ne soit toujours pas devenu ma matière favorite. Je remercie une nouvelle fois mes camarades pour cette belle expérience en leur compagnie.

Pour conclure : Petite rétrospective du projet en images