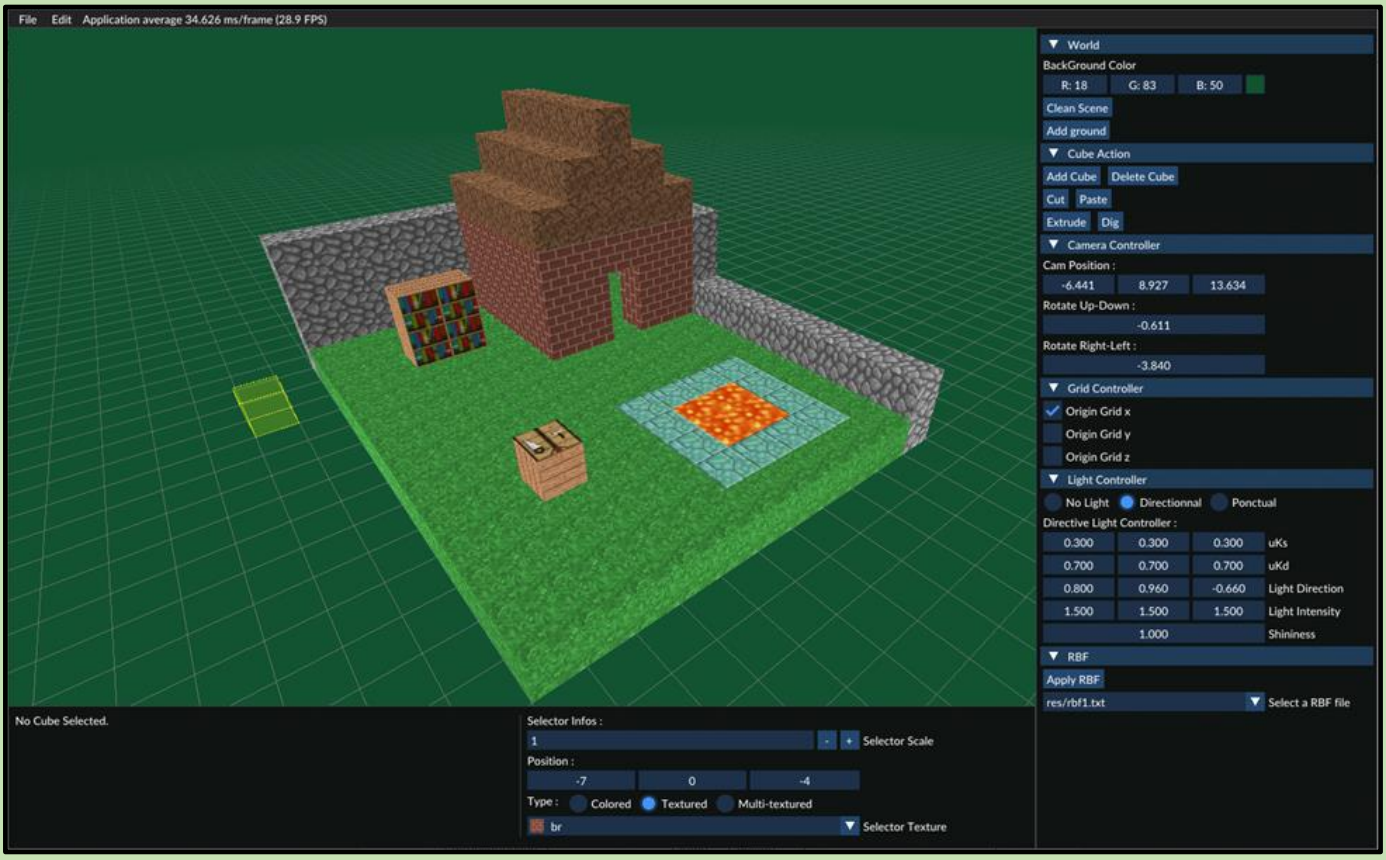
IMAC 2

Minecraft Editor



Antoine Libert

Nicolas Liénart

Sommaire

Fonctionnalités implémentées

Requises		
Nom	Validé	Adaptations et améliorations
Affichage d'une scène avec des cubes	✅	
Edition des cubes	✅	Possibilité de couper-coller les cubes.
Sculpture du terrain	✅	
Génération procédurale	✅	
Ajout de lumières	✅	

Additionnelles		
Nom	Validé	Adaptations et améliorations
Amélioration de la sélection	✅	
Outils de painting	✅	Non amélioré (zone d'application)
Sculpting ++	❌	
Sauvegarde / Chargement de la scène	✅	
Chargement de modèle 3D	❌	
Niveau de discrétisation	❌	
Blocs texturés	✅	Possibilité d'avoir des colorés et texturés en même temps.

Supplémentaires (hors-options)	
Nom	Détails
Ajout de grille	Possibilités d'avoir une grille sur les trois axes de la scène
Dessin par glissement de souris	
Système de mode	L'éditeur envisagé comme une module du projet

Instructions d'utilisation

Compilation et lancement

Les instructions sont disponibles sur le README du repository GitHub :
https://github.com/LibertAntoine/Minecraft_editor/blob/master/README.md

Navigation et raccourcis

Raccourci clavier	*Raccourci souris*	*Accès interface graphique*	*Action*
Camera			
ALT + flèches directionnelles	Click molette + déplacement de la souris.	Onglet Camera Controller -> Cam Position	Déplacement de la caméra horizontalement/verticalement.
ALT + PageUp/PageDown	Défilement molette (scroll)	Onglet Camera Controller -> Cam Position	Zoom/Dezoom de la caméra
CTRL + flèches directionnelles	SHIFT + déplacement de la souris.	Onglet Camera Controller -> Rotate	Rotation de la caméra horizontalement/verticalement.
Sélecteur			
Flèches directionnelles (si focus hors interface ImGui)	Click sur la scène	Onglet Selector Infos -> Position	Déplacement du sélecteur.
		Onglet Selector Infos -> Type	Definie le type de cube sculpté par le selector. En fonction donne accès au choix de la color ou textures(s) que dessine le sélecteur.
Sculpting			
CTRL + I	ALT + click gauche	Onglet Cube Action -> Add Cube	Ajoute un cube à l'emplacement du sélecteur.
CTRL + O	CTRL + click droit	Onglet Cube Action -> Delete Cube	Supprime le cube à l'emplacement du sélecteur.

CTRL + PageUp		Onglet Cube Action -> Extrude	Extrude dans la colonne où se trouve le sélecteur.
CTRL + PageDown		Onglet Cube Action -> Dig	Dig dans la colonne où se trouve le sélecteur.
CTRL + X		Onglet Cube Action -> Cut	Coupe le cube à l'emplacement du sélecteur (s'il existe)
CTRL + V		Onglet Cube Action -> Paste	Colle le cube précedemment coupé (s'il existe).
	ALT + maintien click gauche et déplacement souris		Dessine des cubes sur la trajectoire de la souris.
	ALT + maintien click droit et déplacement souris		Supprime les cubes sur la trajectoire de la souris.
Edition des cubes			
		Onglet Current Cube -> Cube Position	Modifie la position du cube actuellement sélectionné
		Onglet Current Cube -> Type	Modifie le type du cube sélectionné. En fonction ouvre un menu permettant de changer la couleur ou la/les textures des cubes sélectionnés.
	CTRL + maintien click gauche et déplacement de la souris		Painting : modifie les cubes sur la trajectoire de la souris en leur appliquant les presets du sélecteur.
	CTRL + ALT + maintien click gauche et déplacement souris		Painting : modifie les cubes sur la trajectoire de la souris en leur appliquant les presets du sélecteur. Et dessine des cubes seulement au sol
Lumière			
L'onglet Light Controller de l'interface permet de passer entre les différents types de lumières et d'accéder à leur réglage.			

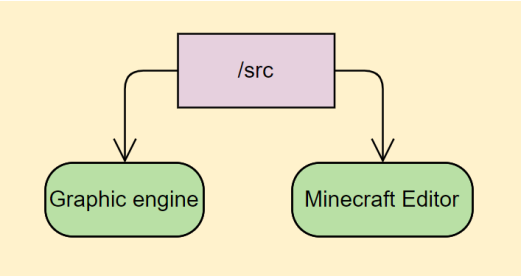
Grille
L'onglet Grid Controller de l'interface permet d'activer/désactiver les grilles sur les différents axes.
RBF
L'onglet RBF de l'interface permet la sélection du fichier de paramétrage et l'application de la RBF.
World
L'onglet World de l'interface permet la modification de la couleur en arrière-plan. Le bouton Clean Scène enlève tous les cubes de la scène. Le bouton Add Ground recrée un sol.

Fichiers RBF

Ce référer à ce readme :
https://github.com/LibertAntoine/Minecraft_editor/blob/master/Doc/README.md

Architecture du projet

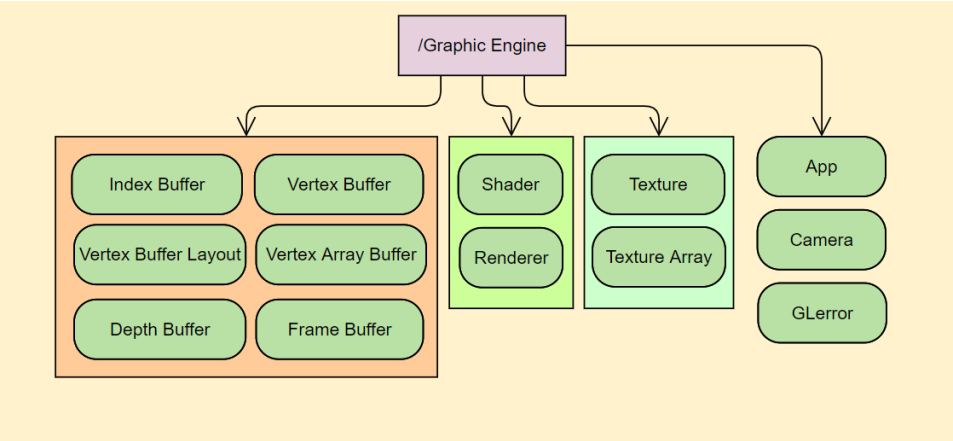
L'architecture de fichier source (/src) se découpe tout d'abord en deux parties majeures : le Graphic engine et l'éditeur (Minecraft Editor).



Le Graphic Engine

La partie Graphic Engine concentre l'ensemble des fonctionnalités qui permettent le rendu d'un environnement 3D. L'ensemble de ces outils sont encapsulés dans des objets, qui en facilite la manipulation et la gestion des erreurs. On y retrouve des objets pour :

- La gestion du contexte de l'application (App) : création du contexte SDL, OpenGL (Glew) et ImGui.
- La gestion des buffers et layouts (VertexBuffer, IndexBuffer, ArrayBuffer, FrameBuffer, DepthBuffer et VertexBufferLayout).
- La gestion des shaders (Shader) : les instructions des shaders (vertex, geometry et fragment) sont récupérées à partir d'un fichier unique comportant l'ensemble de leur contenu.
- La gestion des textures et texture array (Texture et Texture Array).
- La gestion du rendu (Renderer).
- La gestion des cameras (Camera).
- La gestion des erreurs issues des fonctions openGL (GLerror).



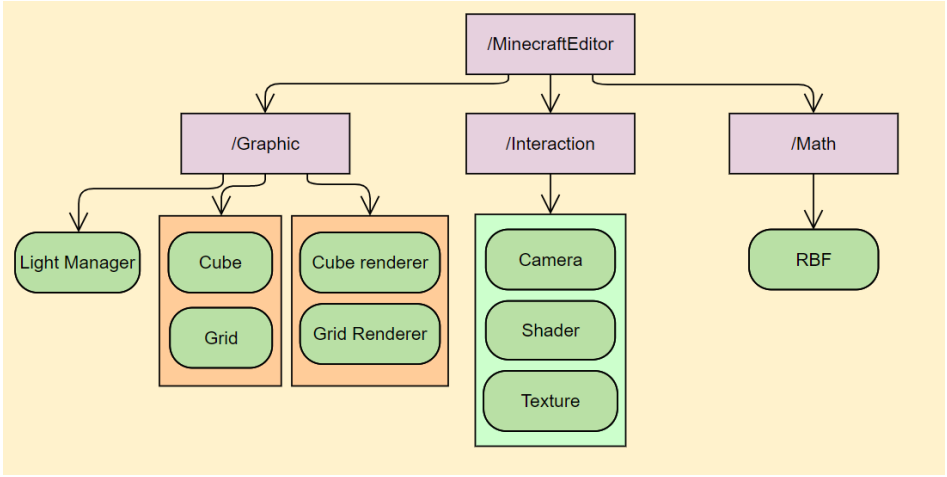
Cette partie se veut générique et autonome. Les fichiers qui la composent ne font pas appel à d'autres fichiers sources du projet et sont facilement implémentables au sein d'un autre projet graphique.

Le Minecraft Editor

La partie Minecraft Editor concentre, elle, l'ensemble de fonctionnalités propres au projet, elle-même subdivisées en trois entités :

- Une partie graphique contenant la logique de dessin et de manipulation des cubes et des grilles, mais aussi des objets gérant leurs rendus graphiques spécifique, et un objet manageant les données relatives aux lumières.
- Une partie interaction contenant la logique relative au curseur et l'édition des cubes. Il contient également un objet pour la gestion de l'interface graphique ImGui et un pour les raccourcis clavier et souris.
- Une partie math contenant les outils de trois Radial Basis Functions.

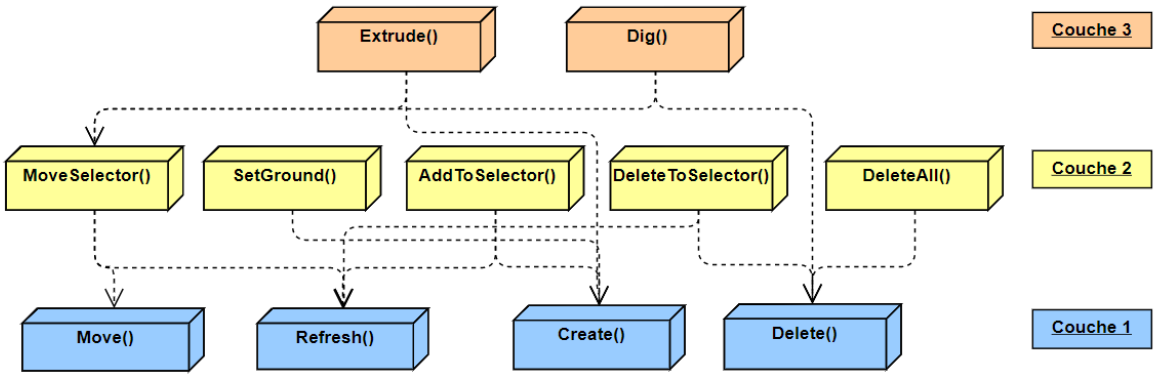
L'ensemble des fichiers de ces deux parties sont gérés par le fichier ModeEditor (dans le dossier Modes) qui se charge de gérer les différentes actions qui s'effectue au lancement de l'éditeur, lors de la phase de rendu de l'environnement 3D, et lors de la phase d'interaction (ImGui et clavier). L'éditeur est considéré comme un mode, un block indépendant au sein de l'application (plus de détail dans le partie Système de Mode).



Les fichiers glsl pour les shaders, les textures et leur proxi, ainsi que les fichiers RBF sont contenus dans le dossier Minecraft/res

Organisation et répartition du projet

Ayant tous les deux déjà travaillé sur le projet Tower Defense au semestre dernier, nous avons rapidement adopté une organisation similaire à celle de ce projet précédent. Nous avons fait tout d'abord au lancement du projet plusieurs réunions pour se mettre au point sur nos connaissances et nos éventuelles difficultés en vues du projet. Puis, nous avons définie l'architecture et les besoins du projet en termes d'objets et de fonctionnalités avant de se mettre à programmer. Nous avons ensuite commencé le développement en fonctionnant par couche d'implémentation : nous définissions les fonctionnalités les plus primaires de l'application, pour tâcher d'appliquer au maximum une logique d'encapsulation. Typiquement après avoir créé l'objet cube et la structure curseur, nous avons créé la logique d'addition/suppression/mouvement, puis la même logique via le curseur, puis la logique d'extrude/dig, chaque fonctionnalité faisant appel à celles des couches précédentes pour éviter toutes redondances de code.



Nous avons ensuite continué l'implémentation du plus simple au plus complexe, en se tenant au courant tous les jours par téléphone des avancés et éventuelles difficultés.

Concernant la répartition du projet à proprement parlé, même si nous sommes tous les deux un peu intervenu sur toutes les parties, Nicolas s'est davantage chargé du sélecteur, en particulier de la sélection au click, de l'outil de painting et sculpting à la souris, et du fonctionnement de la RBF et de la génération procédurale. Antoine s'est occupé de l'affichage des cubes et des grilles, l'édition et le sculpting des cubes, l'interface ImGui et la gestion des lumières. Nous n'avons pas éprouvé de difficulté à travailler ensemble et à communiquer.

Explication exhaustive des fonctionnalités

Fonctionnalités requises

Affichage d'une scène avec des cubes

Fichiers en jeu : Cube, CubeRenderer, res/shader/Cube.shader

L'affichage de la scène repose sur deux éléments primaires : des cubes et des grilles (détaillé dans la partie Grille visuelle)

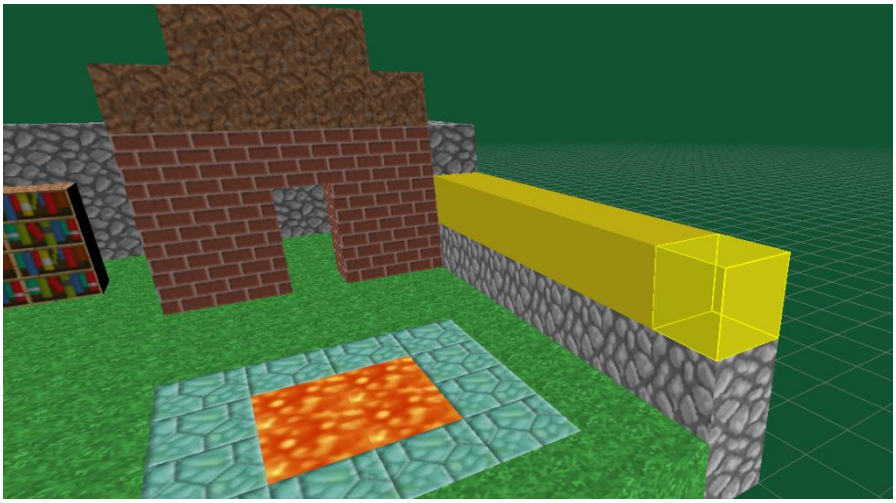
Concernant les cubes, ceux-ci sont à la fois gérés sous la forme d'une liste (std::list) afin de permettre un parcours rapide de l'ensemble des cubes lorsqu'il s'agit de récupérer leurs données en vue de les dessiner, et sous la forme d'un octree de pointeurs vers les cubes de cette liste, afin de gérer spatialement la répartition des cubes et permettre un accès facile à un cube donné via sa position en vue de lui appliquer une modification.

Le dessin de l'ensemble des cubes est effectué à l'aide d'un seul appel de shader, grâce à un appel instancié. Tout d'abord, les informations des différents cubes sont envoyées dans des vertex buffer différents en fonction du type d'information : la position du cube, sa couleur, les id des textures pour chaque face, le type du cube (coloré, texturé ou multi-texturé). Les informations d'un buffer sont actualisées à chaque fois qu'une opération est susceptible d'en impacter les données.

La scène étant constituée d'objets dont la forme est toujours semblable, nous avons fait le choix de définir les sommets constituant les cubes directement au sein du geometry shader. Le geometry shader reçoit donc en entrée les coordonnées d'un seul point, lui spécifiant la position du cube à dessiner. Il se charge pour chaque point reçu de dessiner un cube, ainsi que de fournir les normales, les coordonnées de texture, et d'indiquer la face sur laquelle se trouve chacun des points du cube. Il reçoit en uniform la ModelViewProjection Matrix afin de prendre les données extérieures en compte. Les shaders n'exploitant pas la ModelView et Normal Matrix (les cubes n'ont jamais de rotation), elles ne sont donc pas transmises.

Finalement, le fragment shader récupère le type de cube auquel appartient chaque fragment, ce qui lui permet de déterminer s'il doit lui appliquer une couleur, une texture. Dans le cas de l'application d'une texture, un uniform fournit au fragment shader une texture array contenant l'ensemble des textures pouvant être associées aux cubes, ou leurs faces. Les id fournis par un buffer spécifique permettent de déterminer quelle texture associée à chaque fragment (en fonction de la face où il se trouve).

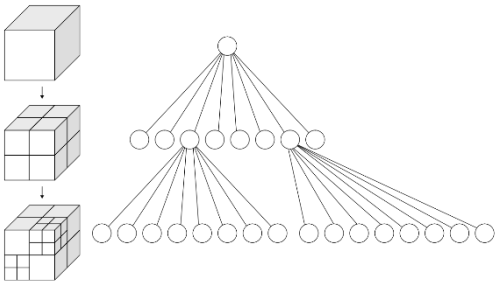
Il est ainsi possible d'avoir des cubes colorés, texturés et multi-texturés au sein de la même scène, sans pour autant multiplier des appels de shader.



Il est par ailleurs possible d'afficher jusqu'à trois grilles dans la scène, toutes les trois centrées en 0 selon l'axe x, y ,ou z de la scène. Cela permet de mieux se repérer dans l'environnement. L'affichage de ces grilles s'effectue avec un pipeline plus classique, comme vu en cours ce semestre (plus d'information dans la partie Grille visuelle).

Enfin, la couleur du fond du monde est également modifiable via l'interface visuelle.

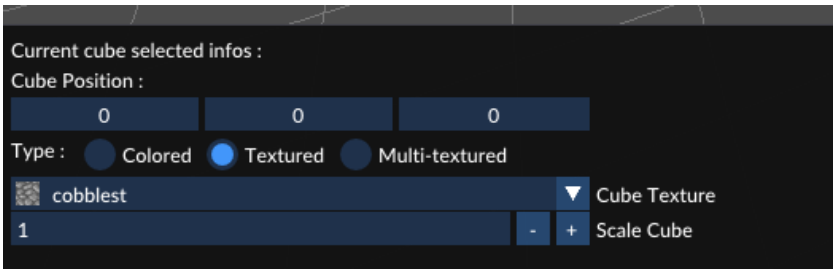
Pourquoi l'octree : L'année dernière notre jeu de tower defence fonctionnait avec un système de cases, pour accéder à un emplacement particulier rapidement grâce à ses coordonnées nous avions utilisé un tableau de pointeurs à côté de la liste chaînée. Cependant nous avons fait le choix d'écarter cette approche dans ce projet car cela aurait requis l'initialisation d'une tableau vide faisant une taille trop importante. Nous aurions pu utiliser un vecteur de la std mais il n'aurait pas permis d'accéder directement à un élément, il aurait fallu itérer à travers l'ensemble des éléments à chaque opération. A la place nous utilisons la structure de donnée octree, géré par la librairie templatisée de Nomis80. L'idée est de subdiviser l'espace en 8 sous-espaces à chaque fois que l'on y place un élément. C'est une généralisation de l'arbre binaire de recherche vu l'année dernière, ensuite pour trouver récupérer un élément particulier on va faire une recherche récursive en partant de la racine. Son homologue 2D, le quadtree, aurait pu être utilisé l'année dernière pour faire un plateau beaucoup plus grand.



Edition des cubes

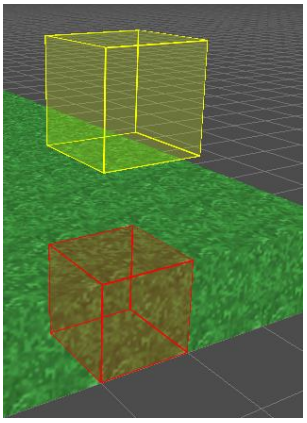
Fichier en jeu : Cube, CubeSelector, Interface, Keyboard

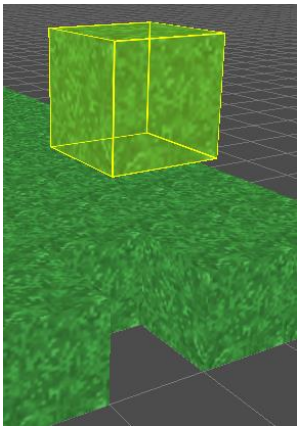
La logique d'édition est centrée autour du curseur, dont l'implémentation est régie par l'objet CubeSelector. Le curseur est dans notre programme une structure indépendante, constituée notamment d'une position, d'un pointeur sur le cube qu'il sélectionne potentiellement et d'un cube particulier, contenant les presets du curseur. Lors de son déplacement, l'élément pointé est actualisé à l'aide d'une recherche dans l'octree de cube. Via le pointeur ainsi récupéré, les informations du cube sélectionné sont affichées sur l'interface et peuvent être directement éditées.



Il est ainsi possible de modifier le type du cube sélectionné, si l'on souhaite le rendre coloré, texturé, ou multi-texturé. En fonction du type actif, le menu permet de modifier la couleur, ou la texture (générale, ou pour chacune des faces) du cube. Il est également possible de modifier la position du cube, soit en modifiant ses données de position, soit en utilisant l'outil couper-coller.

Lorsque l'on souhaite couper un élément, cela change l'état du curseur, un curseur rémanent, d'une autre couleur, s'affiche sur le cube à couper. On peut ensuite naviguer avec le curseur classique pour aller coller le cube ailleurs.





L'affichage du curseur utilise un geometry shader ressemblant à celui utilisé pour les cubes. La position et texture du curseur sont transmises en uniform directement. Le curseur est dessiné sans prendre en compte le Depth Test, ce que permet de toujours visualiser le curseur, même s'il se trouve derrière des cubes.

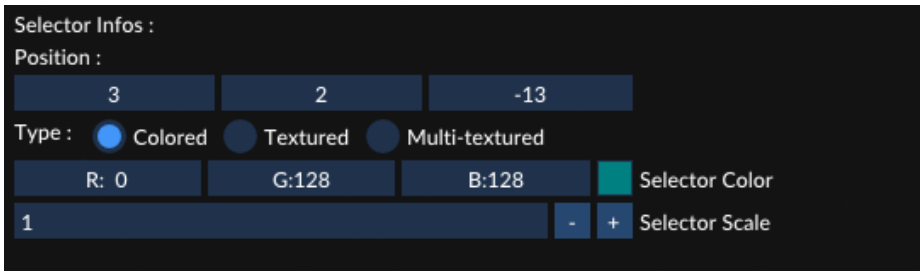
Le curseur est manipulable via les touches du clavier ainsi qu'à la souris (voir la partie Amélioration de la sélection).

Un outil de scale, vise à permettre également de modifier la taille de chacun des cubes indépendamment, mais nous n'avons pas eu le temps de terminer son implémentation.

Sculpture du terrain

Fichier en jeu : Cube, CubeSelector, CubeRenderer, Interface, Keyboard

Pour sculpter le terrain, il est d'abord possible de créer et supprimer un cube via le curseur. Le curseur possède un objet cube, gardant en mémoire les configurations de dessin du curseur. L'interface Infos Selector permet de revenir, comme pour le cube sélectionné, sur la couleur ou texture appliqué par le curseur.

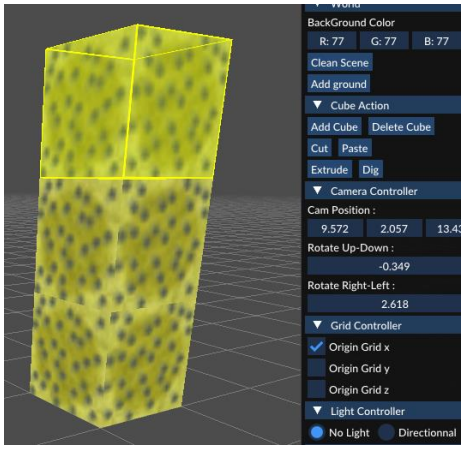


Lorsque que l'on crée un cube via le curseur, cette configuration est récupérée pour créer un nouveau cube, qui sera inséré dans la liste et dans l'octree via sa position. Lors de la suppression via curseur, l'opération inverse s'effectue : le cube est supprimé de la liste puis de l'octree.

L’opération d’extrude consiste d’abord à parcourir la colonne où se trouve le pointeur en partant du haut du monde, jusqu'à tomber sur le cube le plus en hauteur, ou le sol. Ce parcours s’effectue via l’octree. Une fois ce cube identifié, un nouveau cube est créé juste au-dessus, avec la même couleur/texture que le cube identifié. Le curseur est également déplacé sur ce nouveau cube. Si aucun cube n’est présent dans la colonne, le nouveau cube est placé au niveau du sol, avec les configurations en mémoire dans le curseur.

Le dig effectue un processus relativement semblable. Il identifie le cube le plus en hauteur dans la colonne et, s’il existe, le supprime. Le curseur est déplacé sur le cube juste en dessous de celui supprimé, ou sur le sol.

Il est également possible de sculpter et de repeindre le terrain via des outils de sculpting à la souris (voir Outil de Painting). L’onglet World de l’interface permet enfin de vider le contenu du monde et de recréer un sol.



Génération procédurale

Fichiers en jeu : RBF

Le système de génération de terrain basé sur les Radial Basis Functions (RBF) laisse une liberté à l’utilisateur sur les paramètres, cela passe par des fichiers de configuration modifiables au runtime. La classe RBF contient un identifiant sur le fichier à utiliser pour la génération, il peut être changé via l’interface dans la section RBF via le menu déroulant des fichiers disponibles. Lorsque l’utilisateur actionne le bouton d’action le fichier sélectionné est parsé pour en extraire la zone d’application, les points de contrôle avec leur poids respectif, la fonction rbf à utiliser parmi celles proposées, le facteur epsilon utilisé et si l’on souhaite ajouter des probabilités pour des rendus stochastiques.

La zone d’application est définie via deux bornes qui constituent la diagonale d’un cuboid car il n’est pas nécessaire de spécifier chaque sommet de la zone, elle est interprétée grâce à ces informations. Chaque cube qu’elle contient se verra ensuite associé un cube ou un vide.

Les différentes RBF disponibles sont celles spécifiées dans le cours de M. Nozick.

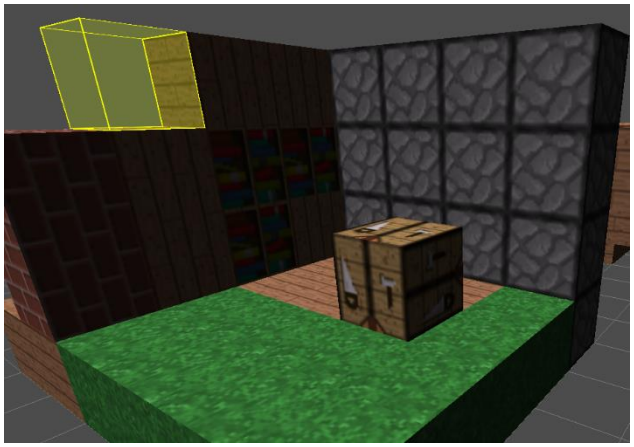
Si le paramètre #noproba est présent dans le fichier alors on place un cube seulement si le scalaire retourné est supérieur à zéro. Sinon on applique des probabilités qui utilisent ce scalaire afin de déterminer la présence d’un cube de façon moins certaine et ainsi créer un terrain accidenté.

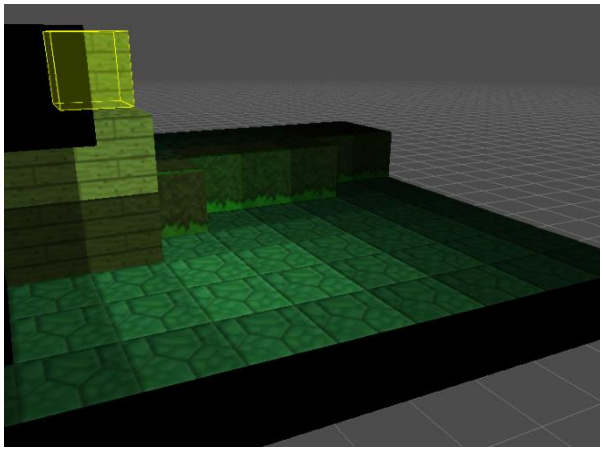
Ajout de lumières

Fichier en jeu : LightManager, CubeRenderer

La scène peut être affichée avec une lumière directionnelle, ou un point de lumière, ou aucune lumière. Pour chacun de ces modes correspond un shaders différent pour le dessin des cubes (la grille et le selecteur ne prenant pas en compte la lumière). Le passage de l'un à l'autre se fait facilement via l'interface, onglet Light Controller. Lors du passage d'un mode à l'autre, l'interface s'adapte pour permettre la modification des différents paramètres lié à la lumière active (coefficients de réflexion, intensité, position ou direction en fonction de la lumière).

L'ensemble de ces paramètres sont transmis au fragment shader via uniform.





Fonctionnalités additionnelles

Amélioration de la sélection

Fichiers en jeu : CubeSelector, Keyboard, CubeRenderer

Nous avons implémenté cette fonctionnalité très tôt car elle nous paraissait utile, ne serait-ce que pour nous aider à tester le programme ou pouvoir débloquer des options supplémentaires.

Lorsque le problème s'est présenté nous avons pensé à utiliser le lancer de rayons et vérifier la première intersection avec un cube ou le sol. On n'aurait certainement pas pensé à la méthode proposée par le sujet. Nous utilisons donc un framebuffer object (FBO) pour stocker les informations de positionnement des cubes et les cases du sol dans une texture. Cette technique rend trivial la vérification des cubes sélectionnables puisque le test de profondeur interne d'OpenGL permet d'écarter automatiquement tous ceux qui sont cachés.

Au départ nous avons voulu stocker les index des cubes mais on est finalement parti pour stocker directement les pointeurs vers les cubes.

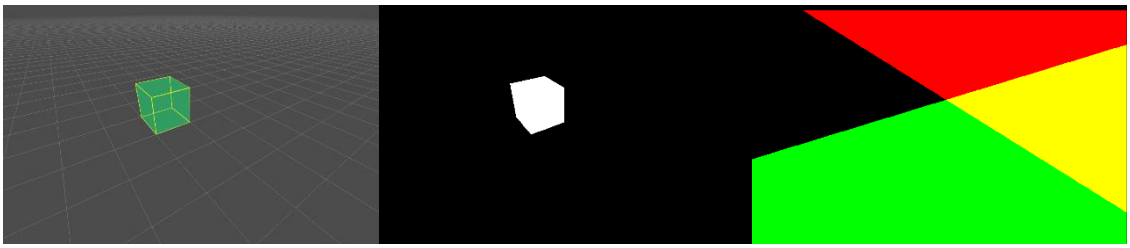
Pour les cubes nous utilisons un shader très similaire à celui destiné à l'affichage normal. En effet les primitives doivent correspondre parfaitement. Sauf que cette fois on transfert et on écrit des données brutes qui ne correspondent pas à des couleurs mais aux pointeurs de chaque cube, ainsi qu'à

un identifiant pour les faces et un booléen explicite sur la présence d'un cube pour prévenir d'éventuelles interpolations de la texture.

Pour la sélection sur le plan du sol nous dessinons une sorte de damier de positions dans la texture. Au lieu de stocker des pointeurs il s'agit maintenant de coordonnées dans l'espace monde, et on utilise aussi la couche alpha comme booléen afin d'indiquer un sol ou le ciel.

Nous utilisons donc une texture pour le sol et une autre pour les cubes. Maintenant lorsqu'il y a un événement de clic on vérifie d'abord la texture des cubes et si aucun n'est sous le curseur on vérifie celle du sol au cas où le curseur est au-dessus d'une case.

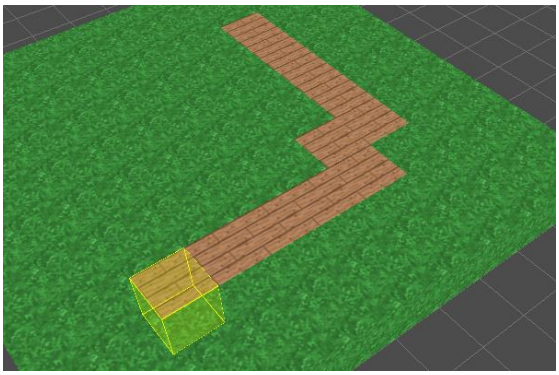
Contrairement à l'année dernière avec le jeu en 2D, la conversion du curseur en 2D vers le monde en 3D n'est pas aussi aisée. Ce procédé d'aplatissement est tout à fait pratique pour permettre la conversion.



Outils de painting

Fichiers en jeu : CubeSelector, Keyboard

On permet à l'utilisateur de changer les propriétés des cubes à la volée en glissant le curseur par dessus afin de donner la même expérience que de la peinture. Pour ce faire on utilise l'outil de sélection à la souris et on utilise l'encapsulation des événements SDL fournie par ImGui pour déterminer le déplacement du curseur. Puis pour chaque cube survolé on va lui appliquer les propriétés courantes du sélecteur.



Blocs texturés

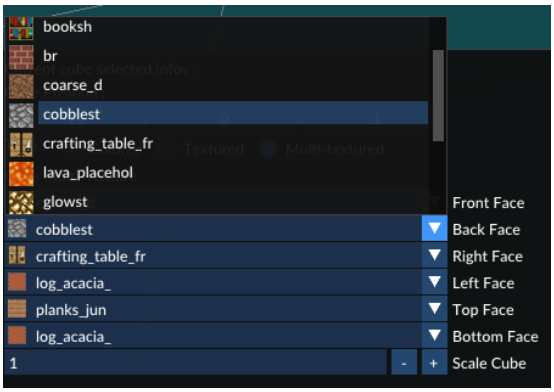
Fichier en jeu : Texture Array, Texture, ModeEditor

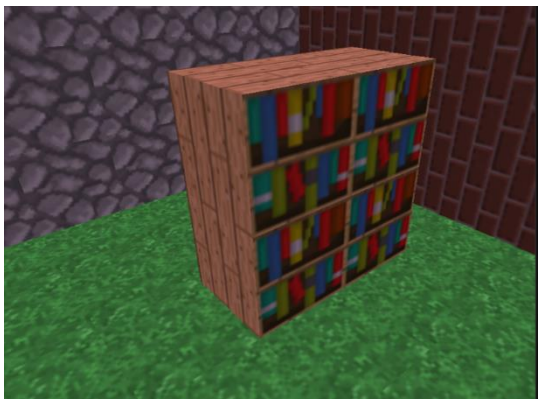
Les textures utilisables pour les blocs sont toutes initialisées à la création du ModeEditor et placées dans un tableau de texture (2D texture array). Les textures y sont listées (en fonction de leur place en profondeur) dans un ordre spécifique.
Lors de la récupération des fichiers de textures, les noms des fichiers sont récupérés, sans leur extension. Ces noms sont associés, à l'aide d'une std::map à un id qui correspond à la place de la texture dans le tableau.

Les cubes ont par ailleurs en attribut un tableau de 6 unsigned int qui leur permettent de garder en mémoire des id des textures pour chacune de leurs faces.

Ainsi, lorsque l'on sélectionne un cube, des menus déroulant dans l'interface nous permettent de visualiser quelles textures sont actuellement appliquées à chacune des faces et de les modifier. Le menu utilise un vecteur de string contenant le nom de toutes les textures et la map pour faire le lien entre ces noms et les identifiants correspondant. Il peut ainsi éditer les id de textures pour les différents cubes.

Nous souhaitions pouvoir prévisualiser les textures dans ces menus déroulants. Or, la librairie ImGui nous permet de venir placer des éléments de texture dans l'interface, mais uniquement à partir de simple texture OpenGL, et non de textures contenues dans un texture array. Nous avons donc opté pour charger une version proxi de chacune des textures. Ces versions portent le même nom que leur équivalant dans le texture array, et un seconde map permet d'accéder à partir de ce nom à un objet texture, permettant de facilement manipuler le proxi.





Chargement et enregistrement de la scène

Fichier en jeu : Interface, Cube Selector.

Pour cela on enregistre les données dans un fichier dont le nom est spécifié par l'utilisateur. Si le fichier existe déjà il est alors écrasé. Lors du chargement, on récupère le fichier spécifié dans le menu déroulant de notre interface. Puis on le parse, chaque cube occupe une ligne. Le menu est rempli dynamiquement par lecture du répertoire "res/scenes/".

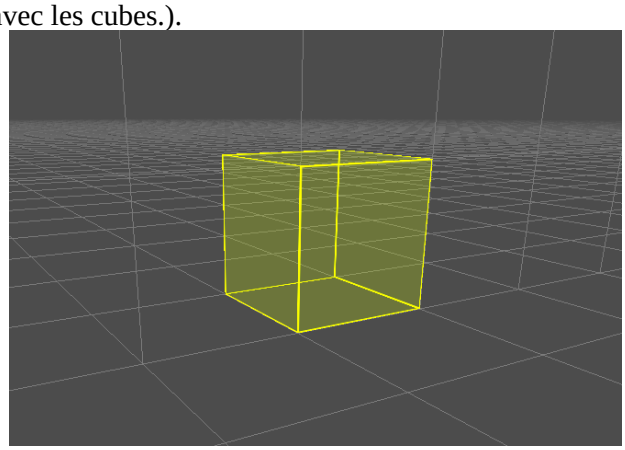
Eléments supplémentaires

Grille visuelle

Fichier en jeu : Grid, GridRenderer

Des grilles peuvent être affiché au niveau des originaux des trois axes de la scène. Ces grilles sont dessinées à l'aide d'une architecture de shaders similaire à celle vu en cours. Les sommets de la grille et les indices de dessin sont d'abord générés par l'objet grille en fonction d'une variable indiquant l'étendue souhaitée pour les grilles. Ces données sont transmises au GPU à l'aide d'un vertex et index buffer. Les grilles sont ensuite dessinées une à une. Une rotation de 90° permet de passer la grille sur un autre axe.

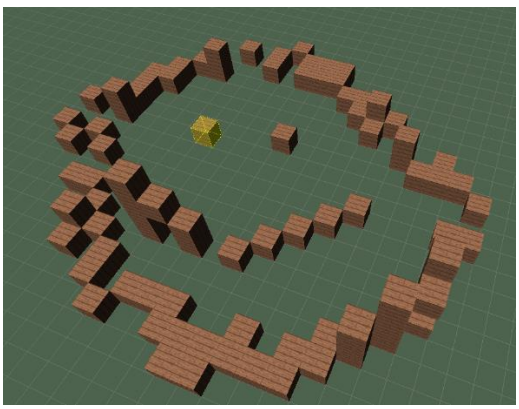
Pour éviter d'avoir la grille trop visible au niveau de l'horizon, les shaders exploitent la position de la caméra. Le fragment shader compare la distance de chaque fragment avec la position de celle-ci et diminue l'indice alpha plus cette distance est grande, ce qui permet de progressivement effacer la grille en arrière-plan. De plus, afin de ne pas devoir dessiner une immense grille si le monde est vaste, chaque grille est centrée sur deux de ses dimensions par rapport à la position de la caméra. Ainsi les grilles "suivent" le déplacement de la caméra (tout en préservant une position avec des entiers, pour éviter un décalage dans l'alignement avec les cubes.).



Dessin par glissement de souris

Fichiers en jeu : CubeSelector, Keyboard

Le procédé est très similaire à l'outil de painting. Cependant cette fois on veut placer un placer des cubes à côté des autres, pour cela il faut décaler le sélecteur par rapport au cube sélectionné par la souris. C'est pour cette raison qu'on a stocké les identifiants des faces dans la texture du FBO, elle nous indique dans quelle face a été cliquée. On fait en sorte que lors d'un dessin glissé on agisse que sur une seule couche par rapport à l'axe y de la hauteur.



Le système de modes.

Fichiers en jeu : Mode, ModeEditor.

Pour la gestion de l'application éditeur, nous avons choisi d'implémenter un système de mode, qui consiste à considère l'éditeur comme un simple partie indépendante du programme. Cette logique, mise en place dans le dossier Modes, consiste à définir une classe abstraite Mode qui contient les méthodes virtuels onEvent, lancé lors de l'application des événements SDL, onRender, pour le rendu graphique, onUpdate, pour l'éventuelle évolution du mode entre chaque image (pas utilisé dans le cas de l'éditeur) et onImGuiRender pour le rendu lié à l'interface graphique.

Ce système est utile à la fois pour le développement et pour la finalité de l'application. Lors de la création de l'éditeur, nous nous sommes servis de différents modes, pour effectuer des tests isolés en parallèle du ModeEditor. Un mode nous permettait également d'accéder facilement à la fenêtre de démo de ImGui pour parcourir les exemples d'interfaces sans avoir à toucher au projet principal. Un ModeMenu, également implémenté dans Mode.h permet de créer une navigation simple entre différents modes chargés dans l'application.

Dans le cas où nous aurions eu besoin d'un menu de démarrage (depuis lequel on aurait pu charger les mondes sauvegardés en les prévisualisant, sans rentrer dans l'éditeur par exemple) ce système aurait pu être utilisé. De même pour constituer facilement une application multi-usage. Même si cette logique n'est pas exploitée dans l'état actuelle de l'éditeur, nous avons souhaité la laisser en place, car nous la jugeons importante pour l'architecture du code, et un outil efficace au développement.

Les maths du projet

Expérimentations des fonctions radiales

Analyse des fonctions radiales sur un graphe :
https://www.geogebra.org/graphing/t9pmh5ud

Captures d'écran de quelques expérimentations :

https://github.com/LibertAntoine/Minecraft_editor/tree/master/Doc/Screenshots/RBF

Pour la RBF nous avons dû utiliser la librairie Eigen pour bénéficier de son optimisation au niveau des décompositions matricielles et de résolution de systèmes linéaires. Nous utilisons d'ailleurs la méthode QR avec les transformations de householder pour privilégier la rapidité à la précision.

Manipulations des données binaires

Pour transférer les données de pointeurs au shader et les écrire dans la texture nous avons été confronté à une incompatibilité de type (voir partie sur les difficultés rencontrées liées aux typage). En effet nos pointeurs sur les cubes étaient codés en 64bit et nous n'avons pas trouvé le moyen de manipuler des données supérieures à 32bit en GLSL, ainsi nous avons dû déconstruire notre pointeur en deux valeurs 32bit que l'on a écrit côte à côte dans le vecteur de couleur du fragment shader. Nous nous sommes basés sur le cours de maths appliqué à l'informatique sur les opérations binaires pour créer cette procédure.

```
GLuint idPart[2];
idPart[0] = (intptr_t(&cube) & 0xFFFFFFFF00000000) >> 32;
idPart[1] = (intptr_t(&cube) & 0xFFFFFFFF);
```

On applique un masque binaire sur le pointeur pour conserver que les informations situées à partir de la puissance 32-1 jusqu'à 64-1, on décale ensuite les données de 32bit puis on cast cela dans un unsigned int OpenGL en 32bit.

Pour la partie de 0 à 32-1 on applique le masque opposé et on fait directement le cast.

Lors de la reconstruction du pointeur après un clic de la souris nous décalons dans l'autre sens les données de la partie gauche puis on applique un masque OU binaire et l'on donne cette adresse à un pointeur sur cube.

Randoms pour les RBF

Fichiers en jeu : RBF, CubeSelector

Il fallait interpréter les scalaires fournis par notre fonction interpolée. On a tout de suite voulu écarter l'approche naïve de placer un cube dès que la valeur retournée est différente de "0". On a préféré l'utiliser comme un facteur de chance sur la probabilité d'obtenir un cube.

Lorsque l'on parse le fichier rbf on conserve le poids le plus élevé et celui le plus bas. Ensuite lorsque l'on interpole les cubes si le scalaire retourné par la fonction est inférieur à 0 on fait en sorte que plus il est proche de la valeur minimale plus il y a de chances de ne pas générer un cube. Dans le cas du scalaire positif on a la même démarche où plus on s'approche de la valeur maximum plus on a de chance de générer un cube cette fois. Pour cela on utilise un générateur de double aléatoire basé sur une distribution uniforme.

L'utilisateur garde la possibilité de désactiver cette fonctionnalité dans le fichier de configuration rbf.

Le C++ du projet

Templates

Fichier en jeu : VertexBufferLayout.h (ligne 42 à 78), VertexBufferLayout.cpp (ligne 4 à 54)

Nous avons eu l'occasion d'implémenter des templates dans la définition des Vertex Buffer Layout. Une layout pouvant fournir des données de différents types au GPU, la méthode permettant d'ajouter une nouvelle entrée dans un layout a été templatisée afin de récupérer le type de données qui va être envoyer dans le vertex buffer associé. La méthode template est ensuite surchargée pour adapter le comportement de la méthode en fonction de type.

```cpp
template<>
GLenum getGLenum<float>() {
    return GL_FLOAT;
}

template<>
GLenum getGLenum<unsigned int>() {
    return GL_UNSIGNED_INT;
}
```

Foncteurs

Afin de changer facilement de RBF et d'expérimenter les fonctionnalités du C++ nous utilisons un pointeur de fonction pour la RBF sélectionnée. En effet nous avons placé chacune des fonctions radiales dans des structs et avons surchargé l'opérateur "( )" pour les utiliser comme fonctions. Nous avons ensuite inclus un objet de type std::fonction et qui contient un "Callable" (cible invocable) prenant un double et renvoyant un double. Cette cible est changeable via des helpers qui attribuent un autre callable à l'attribut fonction.

Fonctions lambda

Fichier en jeu : CubeRenderer.h (ligne 167 à 226)

Les fonctions lambda sont typiquement utilisées lorsque nous avons à parcourir la liste de cube. C'est notamment le cas dans les multiples méthodes de CubeRenderer chargé de mettre à jour le contenu des vertex buffers en fonction de la liste de cubes.

```cpp
std::vector<glm::ivec3> positions;
std::for_each(m_CubeList.begin(), m_CubeList.end(),
    [&positions](Forms::Cube& cube) {
    positions.push_back(cube.position());
});
m_VertexBufferPosition->Update(positions.data(), sizeof(glm::ivec3) * positions.size());
```

Retour d'erreur openGL

Fichier en jeu : GLerror.h, Shader.cpp (ligne 59 à 76)

Le fichier Glerror permet de gérer le report d'erreur lié au fonctions OpenGL et l'arrêt du programme (sur Linux comme sur Windows). Il permet également un retour d'information (la fonction, le fichier et la ligne, ainsi que le message d'erreur).

Le fichier Shader s'occupe lui aussi, lors de la compilation des shaders, de retourner dans la console les informations liées à d'éventuelles erreurs des compilations, spécifiant le shader concernant, la ligne, le message.

Vérification des fuites de mémoire avec Valgrind

Vers la fin du projet nous avons analysé les fuites de mémoire du programme au travers de Valgrind, nous avons corrigé les problèmes grâce aux indications fournies en retour. Cependant il reste encore des fuites mais impossible pour nous de remonter à la source puisque le backtrace de Valgrind pointe sur des fichiers de la glibc comme source de leak du type (dlopen_doit (dlopen.c)). D'après nos recherches il ne s'agirait pas d'un problème venant de notre programme. Affaire à suivre.

Les points d'amélioration du projet

Optimisation rendering :

Bien que nous ayons toujours voulu optimiser notre code au fur et à mesure, notre éditeur souffre de chutes de performance lorsque la scène présente plusieurs douzaines de cubes. Cela limite grandement les possibilités d'expérimentation de la génération procédurale de terrain.

On a implémenté le rendu via les geometry shaders pour ne plus faire un draw call par cube à dessiner mais profiter du rendu instancié car nos cubes sont des géométries simples mais similaires.

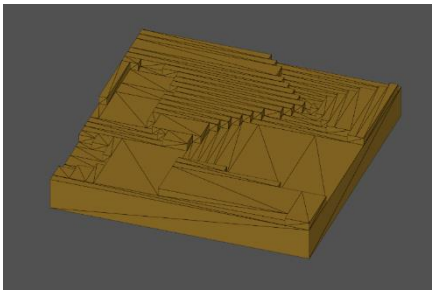
Le backface culling qui évite le rendu des faces non visibles des géométries nous a bien aidé pour optimiser les rendus. Il reste néanmoins un gros travail à faire pour avoir un résultat satisfaisant, voici ce à nous avons pu penser peut penser :

Fusion des formes

Ne pas rendre les faces collées entre elles car elles sont forcément cachées, pour cela il faudrait faire des rendus en utilisant des faces comme primitives. Les économies croient à échelle logarithmique. Dès lors que l'on rend un bloc de 4 cubes collés formant un plus gros cube on retire 33% des triangles, 50% pour un bloc 3 x 3 x 3, etc.

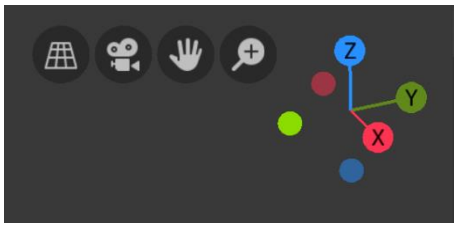
Greedy mesh

Une méthode encore plus efficace serait d'utiliser un greedy meshing, comme le jeu Minecraft. L'idée est de dessiner tout ensemble de cubes formant un cuboid comme étant une seule géométrie : soit 12 triangles au total. On y gagne une réduction exponentielle du nombre de triangles à dessiner en fonction de la taille du bloc. Cette stratégie est en plus compatible avec la technique précédente de fusion.



Meilleure interaction pour modéliser (plus intuitive)

On aurait aimé proposer des outils visuels pour faciliter le déplacement dans l'éditeur. On pense notamment à une girouette présentant les axes comme de nombreux logiciels de modelling 3D. On envisagerait également un joystick de déplacement de la caméra par rapport à son propre repère. Et un slider modifiant la vélocité du zoom/dézoom qui revient en position lorsqu'il est relaché, nous n'avons pas trouvé cette possibilité dans ImGui.



Les difficultés rencontrées

Des blocages au niveau des types de données entre le CPU et le GPU

À de nombreuse reprise on a été confronté à des erreurs de compilation de shaders et des erreurs numériques à cause de mauvais transferts et conversions des données typées entre le C++ et la carte graphique. Comme nous avons tenu à utiliser le type adapté à chaque usage on a utilisé les int, float et unsigned int avec leurs équivalents vec, ivec et uvec. L'un des problèmes était que opengl passe les données en float en transférant du CPU au GPU il faut utiliser glVertexAttribIPointer (avec un "i" majuscule) pour forcer l'usage de valeurs entières. -> http://docs.gl/gl3/glVertexAttribPointer c'était à l'origine d'erreurs impardonnables dans le code GLSL.

On avait aussi la gestion du format de données au niveau des textures et au début c'était assez déroutant. Entre le format interne, le format tout court et le type. -> http://docs.gl/gl3/glTexImage2D

À ce niveau là OpenGL requiert une bonne rigueur de la part du développeur.

Une petite anecdote : lorsque l'on bossait sur les textures du framebuffer de sélection on est resté plusieurs heures bloquées sur la récupération des données avec glReadPixels. On récupérait des valeurs nulles et on fait le tour d'Internet pour résoudre le problème. Il aura fallu essayer une combinaison avec comme format GL_RGBA_INT et ça a fonctionné. La documentation d'OpenGL ne mentionnait pas cet enum particulier parmis les options autorisées. En fait on a découvert qu'il était propre à GLEW. On a donc appris à remettre en cause la documentation et que l'intuition du programmateur compte aussi.

http://docs.gl/gl4/glReadPixels

Difficultés pour optimiser le pipeline graphique

Nous avons eu des difficultés pour déterminer de quelle façon nous pouvions optimiser le rendu des cubes. Nous avons tout d'abord appliqué la méthode que nous avons vu en cours, dessinant les cubes un à un avec une fonction lambda, en transmettant les informations de couleur et texture en uniform. Une méthode similaire à celle actuelle pour les grilles.

Toutefois, voyant que l'ajout de cube provoquaient rapidement des ralentissements, nous avons voulu trouver des solutions de rendu plus rapides, notamment lorsque le nombre de cube est important. Nous avons donc fait des recherches parmi une documentation très riche sur internet sur le sujet, découvrant des techniques dépassant nos connaissances actuelles. Beaucoup d'exemples présentés suggéraient dans le cas du dessin d'une forme toujours similaire, d'utiliser un geometry shader dessinant la forme à partir d'un point, ou de faire de l'instancing, mais dans des situations différentes de la nôtre (pas d'édition des formes au cours du temps…). Nous n'avions jamais manipulé ni geometry shader, ni d'instanciation avant le début de ce projet, et dans le temps que nous disposions, il nous a été difficile de déterminer si l'application d'une méthode ou d'une autre pouvait être bénéfique à notre situation (car elles semblaient efficaces uniquement dans certains cas, voir qu'avec certaines cartes graphiques…).
Nous avons toutefois cherché à mettre en place le pipeline graphique décrit au début de ce dossier, sans toujours savoir aujourd'hui si d'autres formes d'optimisation aurait été plus souhaitable. Dans tous les cas, cette utilisation plus avancés des shaders nous a énormément appris sur la manipulation des données entre CPU et GPU comme détaillé précédemment, mais également sur le passage de données entre les shaders eux-mêmes (données varying, flat...), et sur le fonctionnement et les possibilités offertes par le geometry shader notamment. Nous sommes donc très satisfaits d'avoir effectué ses recherches et d'avoir au sein de ce projet pu les mettre en application, même si nous ne sommes pas arrivés à une optimisation très satisfaisante.

RBF

Nous avons eu des difficultés à maîtriser et prédire les réactions de la RBF, comme plusieurs paramètres entrent en jeu dans les différentes fonctions radiales. Mais au fur et à mesure de son utilisation et d'essais avec différents réglages nous sommes parvenus à générer des formes prédictibles via les fichiers de configuration.

Parties individuelles

Antoine Libert

J'ai pour ma part le sentiment d'avoir beaucoup appris de la réalisation de ce projet. Toutefois, j'ai parfois éprouvé la frustration de ne pas pouvoir, faute de temps, autant me renseigner que je l'aurais voulu autour de certaines implémentations. Ce projet était vaste, demandant un grand nombre d'implémentation, ce qui donnait parfois le sentiment de ne pouvoir que survoler certaines technologies que l'on découvrait. Je l'ai particulièrement ressenti concernant la création d'interface avec ImGui, l'optimisation des shaders, ou encore les manières de gérer les textures (et matériaux) avec OpenGL, où j'aurais aimé avoir davantage de temps pour expérimenter ces notions, même si je comprends par ailleurs qu'elles ne rentraient pas dans le cœur de ce que l'on attendait de nous pour ce projet.

Je n'en reste pas moins satisfait du travail réalisé. Ce projet m'a aussi permis d'acquérir une certaine aisance de programmation en C++. Appuyer par les cours de ce semestre, j'ai le sentiment de plus facilement abstraire les problématiques de code rencontrées, et d'identifier plus rapidement les outils de programmation me permettant d'obtenir les résultats souhaités. Ayant également réalisé le Tower Défense avec Nicolas le semestre dernier, nous avons pu éprouver notre progression entre ces deux projets, autant en termes de compréhension de l'OpenGL et C/C++ que d'efficacité.

Nicolas Liénart

Durant ce projet j'ai pu grandement profiter de RenderDoc, un outil de débogage graphique conseillé par Guillaume Haerinck. Il me permettait de trouver les problèmes dans la chaîne de rendu et m'a beaucoup aidé à analyser les textures de rendu offscreen du FBO pour la gestion de la l'interaction

et vérifier les données de pixels, la correspondance avec le rendu normal des cubes, etc. Au niveau du débogage C++ j'ai continué à utiliser GDB et à en apprendre toujours plus, à connaître ce genre d'outils bas niveau généralement utilisés comme API je gagne une certaine indépendance vis à vis des IDEs et de la machine que j'utilise. D'ailleurs j'ai grandement amélioré ma configuration avec Vim, je dispose de fonctionnalités avancées notamment grâce à l'usage du Language Server Protocol (LSP) et d'un plugin permettant de s'y connecter.

Au niveau de mon workflow git j'ai expérimenté le rebase tout au long du projet, quand je travaille sur une fonctionnalité je fais une branche locale, je fais mes commits dessus, puis je rebase par rapport au master GitHub avant de faire un merge de ma branche dessus. Ainsi je pouvais résoudre les conflits au fur et à mesure. J'ai de façon générale grandement amélioré mes connaissances avec Git et je suis à l'aise avec l'outil maintenant, cette compétence sera particulièrement utile pendant le projet tuteuré. La prochaine étape sera de découvrir le workflow d'intégration avec les pull requests et des repository propres à chaque collaborateur.

Concernant le projet en lui-même j'ai beaucoup apprécié pouvoir faire le parallèle avec celui de conception 3D de M. De Robillard. En effet avoir la vision de l'outil d'un côté et l'utilisateur de l'autre permet une certaine attention aux détails et on fait mieux les choses, on pense à l'optimisation, l'expérience utilisateur, etc. Cela me fait penser au back-end/front-end en web.

Je note également que la manipulation de langage bas niveau est toujours très intéressant pour mieux coder en haut niveau, on adopte une certaine "empathie" pour la machine si je puis dire.

Enfin ce projet fut riche en challenge, et j'ai beaucoup apprécié me confronter aux problèmes et la recherche de solutions pour les résoudre.