Solène Mary-Vallée
Baptiste Ory

Éditeur-visualisateur de terrains et scènes 3D ImacraftSB

/Projet prog. IMAC2 S3

Synthèse d'images, Programmation objet et Maths-info
/ Venceslas Biri, Céline Noël et Vincent Nozick

/Sommaire

/Introduction

/Présentation du projet

Le projet qui nous a été confié pour les cours de « Synthèse d'images », « Programmation objet » et « Mathématiques pour l'informatique » était la **réalisation d'un programme d'édition et visualisation de terrains et scènes 3D en langage C++**.

Ce programme devait pouvoir **générer des scènes constituées uniquement de cubes et permettre à l'utilisateur de modifier, ajouter ou supprimer des cubes à l'aide d'un curseur et d'une interface**. La scène devait être **en 3D, parcourable à l'aide d'une caméra et rendue avec des lumières**. C'est ce que peut faire très basiquement un logiciel de modélisation comme **Blender** ou encore le célèbre jeu vidéo **Minecraft**.

On a choisi d'appeler notre programme **ImacraftSB** en référence à la formation, au jeu Minecraft et à nos initiales.

Le lien **GitHub** du projet est : https://github.com/SolHaine/ProjetProg_ImacraftSB. Par ailleurs, nous nous sommes organisés grâce à un **Trello**, dont voici le lien : https://trello.com/b/NMpiYTXl/projet-prog-imacraftsb.

/Mise en contexte

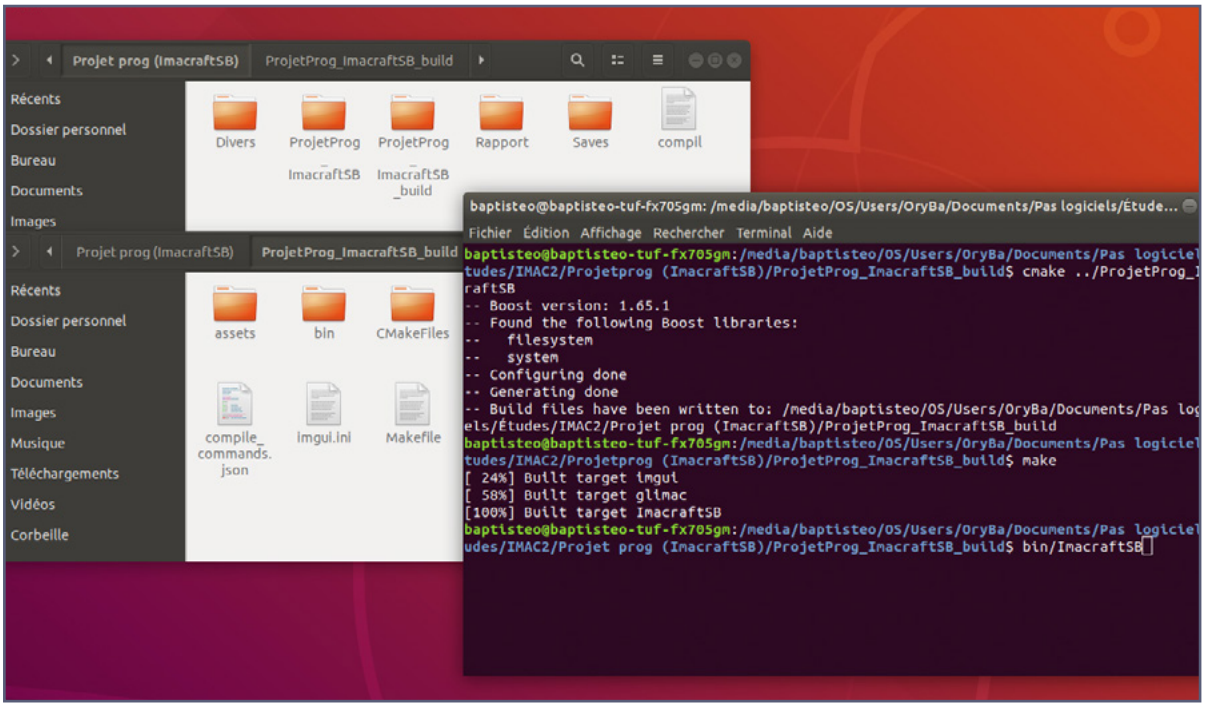
Voici la **structure des répertoires** du projet :

ProjetProg_ImacraftSB/
- **\-- assets/textures/ ->** textures des cubes
- **\-- CMake/et CMakeLists.txt ->** compilation (grâce à CMake)
- **\-- doc/->** documentation (sujet et rapport)
- **\-- include/ ->** fichiers .hpp
- **\-- lib/ ->** librairies (glimac, glm et imgui).
  À noter que les autres librairies nécessaires au programme (OpenGL, SDL2, GLEW, Eigen3 et Boost) doivent avoir été installées sur la machine de l'utilisateur
- **\-- src/->** fichiers .cpp, shaders (vertex et fragment shaders) et exemples de scènes
- **README.md ->** lisez-moi

Pour tester le programme : placez-vous dans un dossier de build dans le même répertoire que le projet (ex : ProjetProg_ImacraftSB_build), créez le build avec CMake, compilez et lancez le programme :

```
cmake ../ProjetProg_ImacraftSB
make
bin/ImacraftSB
```

L'exécutable est donc dans le dossier /bin du dossier de build. Les scènes sauvegardées sont dans /bin/savedScenes. Lisez le fichier README.md (à la racine du projet) pour plus d'informations.



Mise en contexte du fonctionnement du programme sur Linux (Ubuntu)

/Règles

Une **scène se génère aléatoirement** (grâce aux « radial basic functions ») à partir de quatre niveaux de cubes (un « bedrock », deux « cobblestone » et un « grass »). L'utilisateur peut alors se **déplacer dans la scène avec une caméra libre**. Il peut aussi **modifier la scène** grâce aux fenêtres et boutons de l'interface : ajouter des cubes avec des couleurs et des textures, détruire des cubes, extruder, creuser, ajouter/supprimer une couleur, ajouter/supprimer une texture, passer en mode nuit, ajouter/supprimer une lumière ponctuelle, réinitialiser les lumières, générer un nouveau monde aléatoire (grâce aux « radial basic functions ») et générer un nouveau monde plat. S'il le souhaite, grâce à une barre de menu, l'utilisateur peut **sauvegarder sa scène ou charger une de ses scènes enregistrées**.

Voici les **commandes utilisateur** du jeu :

> Echap : quitter

> Caméra :

> Z / Q / S / D : déplacer la caméra horizontalement et verticalement

> A / E : déplacer la caméra en hauteur

> Clic gauche + glisser : tourner la caméra

> Curseur :

> O / K / L / M : déplacer le curseur horizontalement et verticalement

> I / P : déplacer le curseur en hauteur

> Interface :

> Fenêtre bas-droit (Toolbox) : ajouter/supprimer un cube, extruder/creuser, choisir une couleur pour le prochain cube ajouté, ajouter/supprimer la couleur du cube sélectionné, ajouter/supprimer la texture du cube sélectionné

> Fenêtre bas-gauche (Texturebox) : choisir une texture pour le prochain cube ajouté

> Fenêtre haut-droit (Lightbox) : changer entre le jour et la nuit, ajouter/supprimer des lumières ponctuelles, réinitialiser les lumières

> Fenêtre haut-droit 2 (Worldbox) : générer un nouveau monde aléatoire, générer un nouveau monde plat

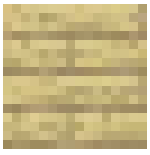
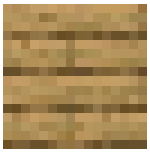
> Barre de menu (en haut) : sauvegarder/charger une scène (dans/bin/savedScenes)

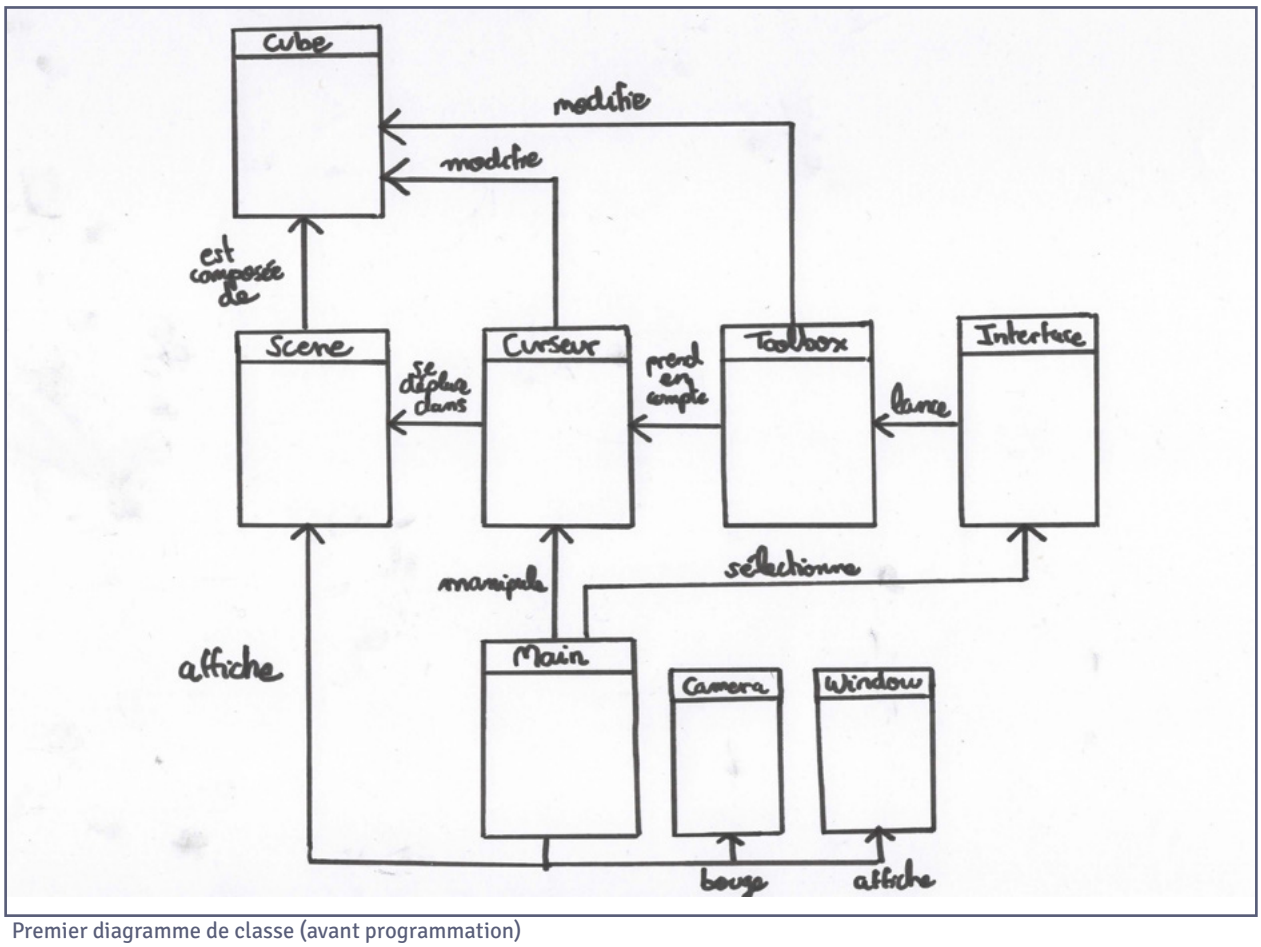
/Structure et algorithme

/Fonctionnalités

Fonctionnalités	Description
Création d'un cube	Modéliser un cube en 3D : Cube.cpp et shaders
Création d'une scène	Générer plusieurs cubes dans une scène, avec chacun sa spécificité (position dans la scène, couleur, etc.) : Scene.cpp et shaders
Visualisation de la scène	Affichage de la scène sur un écran : View.cpp et main.cpp
Parcours de la scène avec la caméra	Déplacer une caméra dans tous les sens pour visualiser la scène : FreeFlyCamera.cpp
Édition des cubes	Déplacer un curseur (comme la caméra) permettant de sélectionner les cubes de la scène : Cursor.cpp et shaders
Sculpture du terrain	À partir du curseur, pouvoir choisir et modifier la couleur des cubes de la scène (add and delete color), ajouter et supprimer de cubes (create and delete), extruder et creuser (extrude and dig) : Scene.cpp et Interface.cpp
Génération procédurale	Générer des terrains aléatoires avec les « radial basis functions » : Math.cpp, Scene.cpp et Interface.cpp
Lumières	Éclairer la scène, pouvoir ajouter et détruire des lumières ponctuelles : Light.cpp, Interface.cpp et shaders

Fonctionnalités supplémentaires	Description
Sauvegarde/chargement de la scène	Sauvegarde une scène créée par l'utilisateur et charger une scène (sauvegardée préalablement par exemple) : Interface.cpp et Scene.cpp
Blocs texturés	En plus des couleurs, pouvoir choisir et modifier la texture des cubes : Scene.cpp, Interface.cpp, Texture.cpp et shaders



Premier diagramme de classe (avant programmation)

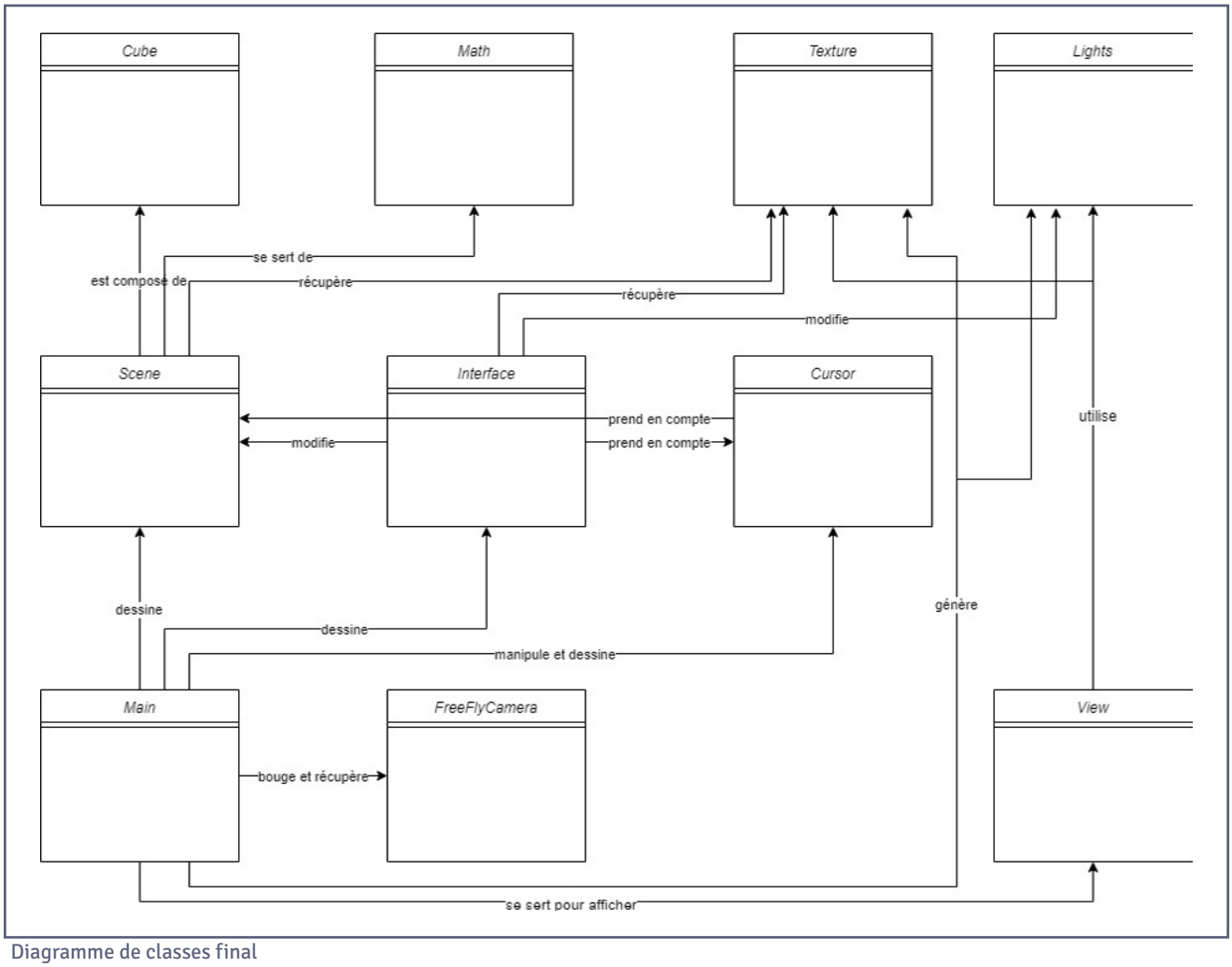


Diagramme de classes final

/Fonctionnalités générales

/Affichage d'une scène

Pour parvenir à afficher une scène, nous avons d'abord défini **la structure d'un cube** dans Cube.cpp (position de ses vertex et normales), afin d'avoir un affichage cohérent pour un seul cube lors du dessin. Nous avons pour cela utilisé un VBO, un IBO et un VAO (OpenGL).

Une fois que ceci était fait, nous avons défini **la structure d'une scène**. Pour cela, on a **réutilisé le code de la structure d'un seul cube et créé un vecteur des positions de tous les cubes et de leurs caractéristiques** (couleurs et plus tard textures) dans Scene.cpp (s_vertices). Ainsi, toute la scène (assemblage de tous les cubes) pouvait être **dessinée en une seule fois**. Nous avons pour cela utilisé un VBO, un VAO et repris le VAO du cube (OpenGL). Dans un premier temps (avant les « radial basic functions » et la génération du terrain aléatoire), en utilisant des largeurs et hauteurs choisies à la construction de la scène, nous avons créé un monde plat.

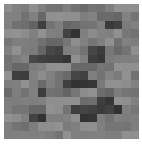
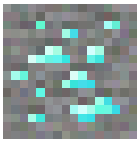
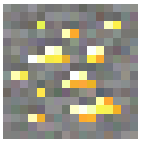
Pour le parcours de la scène avec une caméra, nous avons choisi de coder une **caméra « libre »** dans FreeFlyCamera.cpp, manipulable grâce à des commandes utilisateur dans main.cpp. Ainsi, la scène est visualisable et parcourable avec un assemblage de matrices (View Matrix de la caméra, Projection Matrix et Model Matrix) : voir main.cpp, View.cpp et les shaders.

/Édition des cubes

Visuellement, le curseur ressemble à une **armature de cube** : on voit les arêtes. Il est **créé de façon similaire au cube** dans Curseur.cpp, à la différence que les primitives tracées ne sont plus des triangles, mais des **lignes** (cela modifie les valeurs envoyées dans l'IBO). Par contre, curseur et cube utilisent les mêmes shaders.

Dans la boucle de rendu, nous avons **désactivé le calcul de profondeur** afin de rendre visible toutes les arêtes du curseur, mais également — et surtout — pour pouvoir voir le curseur même s'il est derrière un cube (sans cela, le curseur serait caché par ce dernier).

Le curseur possède également un ensemble de **méthodes nécessaires à son déplacement dans la scène**. Celles-ci étant par la suite exécutées dans la boucle de rendu (main.cpp). De plus, nous avons choisi de **modifier la couleur du curseur selon s'il se situe sur un cube où non** : il est rose lorsqu'il se trouve sur l'emplacement d'un cube, bleu sinon. Le curseur connaît donc Scene.cpp.

/Sculpture du terrain

Toutes les fonctions relatives à **la sculpture et la modification du terrain** — et donc des cubes — ont été codées dans Scene.cpp. En effet, dans un souci de **forte cohésion/faible couplage**, nous trouvions qu'il était préférable de ne pas trop tout séparer afin d'atteindre plus facilement certaines fonctions. Ces **fonctions agissent sur le tableau des cubes de la scène** (évoqué dans « affichage d'une scène ») en ajoutant et retirant les cubes du tableau, ou en modifiant leurs particularités. Ensuite, elles **mettent à jour le VBO** de la scène pour que les changements soient pris en compte au prochain dessin.

Il a été assez difficile de coder l'extrusion et le creusage (« extrude » et « dig »), car nous voulions que cela soit rapide, propre et cohérent. On ne voulait pas parcourir plusieurs fois le tableau des cubes par exemple. Mais, en codant à part une fonction qui va chercher le plus haut cube de la colonne, nous sommes arrivés au résultat escompté.

Puis, pour utiliser ces fonctions, nous avons créé une **interface grâce à la librairie imgui** : Interface.cpp. Les fonctions de sculpture du terrain sont donc exécutées dans l'interface (avec la **fenêtre « Toolbox »**) en récupérant la position du curseur et/ou des caractéristiques choisies par l'utilisateur dans l'interface (une couleur par exemple).

/Génération procédurale

Pour la génération procédurale, nous avons découpé cela en plusieurs fonctions.

Tout d'abord, nous avons créé dans Math.hpp une **structure contenant toutes les informations nécessaires à l'interpolation avec les « radial basis functions »** : points de contrôle, poids associés aux points de contrôle, coefficients (omégas : vide au départ) et nombre de points de contrôle — car nous générons aléatoirement points de contrôle et poids.

Ensuite, Math.cpp rassemble **la fonction d'interpolation et toutes les fonctions nécessaires à sa création**. Nous avons donc **phi : la fonction radiale**. Changer le comportement de phi permet de jouer sur le mode d'interpolation (linéaire, multiquadratique, inverse quadratique, gaussienne), sans pour autant modifier le fonctionnement du reste. Et, nous avons deux autres fonctions : l'une servant à **résoudre l'équation matricielle permettant de remplir les omégas**, l'autre à **obtenir des points de contrôle et poids aléatoires**. Comme en pratique nous n'utilisons pas un grand nombre de points de contrôle, nous pouvons facilement résoudre le système en inversant la matrice (si les dimensions sont trop grandes, cette méthode n'est pas très efficace). Enfin, grâce à toutes ces fonctions, nous avons pu coder la **fonction d'interpolation** qui peut donner un poids pour n'importe quelle position.

À présent, ce qu'il nous faut comprendre c'est ce à quoi correspondent les poids. Dans notre cas, **le poids correspond à « l'altitude » du terrain**. Si le poids est positif, le terrain sera plus haut et s'il est négatif, il sera plus bas. La génération d'un tel terrain se fait dans Scene.cpp. Nous partons avec pour base le terrain plat, nous calculons pour une grille 2D de la taille du terrain les poids avec la fonction d'interpolation et nous utilisons la fonction « dig() » ou la fonction « extrude() » un certain nombre de fois, selon la valeur et le signe du poids.

Dans l'interface, la **fenêtre « Worldbox »** permet de générer un nouveau monde aléatoirement ou de repartir sur un terrain plat (Interface.cpp).

Voir la partie **« Radial basic functions »** pour une analyse précise des différentes fonctions.

/Lumières

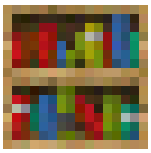
Le comportement des lumières est géré dans le fragment shader où l'on trouve **deux fonctions : une pour la lumière directionnelle (le soleil) et une pour les lumières ponctuelles**. Elles modifient la couleur du cube en fonction de sa position et de la normale de chaque face.

Light.cpp permet de gérer **le nombre et la position des lumières ponctuelles, ainsi que l'état de la lumière directionnelle (jour/nuit)**. Ces informations sont envoyées au fragment shader pour qu'il puisse calculer l'influence de chacune des sources lumineuses.

Le calcul des lumières étant assez lourd, plus on rajoute de points lumineux, plus le rendu est lent. Cela se ressent lorsque l'on veut se déplacer dans la scène. Nous avons ajouté quelques **restrictions à l'ajout d'une nouvelle lumière** : il ne peut y avoir deux lumières ponctuelles au même endroit et le nombre maximal de lumières ponctuelles est limité à 100 (qui ralentit tout de même beaucoup le programme). Les **intensités lumineuses des lumières, ainsi que la direction du soleil (lumière directionnelle), ont été fixées arbitrairement** dans le fragment shader. Cependant, nous aurions pu imaginer que l'utilisateur puisse jouer dessus.

Finalement, nous avons placé en plus une **lumière ambiante**, faisant en sorte que les faces ne soient pas toutes noires si elles ne reçoivent pas de lumière. Le mode nuit correspond à une autre lumière ambiante.

Dans l'interface, la **« Lightbox »** permet de choisir le mode jour/nuit, placer une nouvelle lumière ponctuelle à l'emplacement du curseur ou retirer toutes les sources de lumière ponctuelles (Interface.cpp).



/Fonctionnalités additionnelles

/Sauvegarde et chargement de la scène

Pour sauvegarder la scène, nous avons créé dans Scene.cpp une fonction qui **écrit dans un document les informations nécessaires à la recréation d'une même scène** : positions, couleurs et textures des cubes, ainsi que les lumières.

Pour la **sauvegarde**, toutes les informations dont on a besoin sont donc dans le **vecteur des cubes de la scène (s_vertices)**, mais qui est un vecteur de la structure ShapeVertexScene. Il faut donc décomposer chacun des éléments du vecteur. Nous devons donc écrire sur le document de sauvegarde, chacun leur tour, les positions (x, y, z), couleurs (r, g, b), puis textures de tous les cubes constituant la scène. Les lumières fonctionnent sur le même principe.

Pour la **lecture**, nous remplissons les données de la scène selon le même raisonnement. On peut manipuler les fonctions de sauvegarde et de chargement dans la **barre de menu de l'interface** (Interface.cpp).

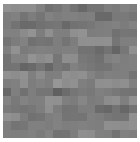
/Blocs texturés

Les textures ont été un gros défi étant donné que nous avions choisi de dessiner toute la scène d'une traite au départ, et non cube par cube. Pour coder les textures, il a donc fallu utiliser un **tableau de samplers** dans le fragment shader (remplis dans View.cpp) et faire une **boucle sur toutes les textures à utiliser avant le dessin** (dans Scene.cpp).

Il a aussi été compliqué de réfléchir à un système simple et propre qui permettrait de mettre six textures par cube. Heureusement, les « **cubemaps** » ont répondu à notre problème. Une « cubemap » est une texture qui contient six textures 2D individuelles applicables à un cube, et ce seulement grâce à sa position. Il n'y avait donc pas besoin de s'embêter avec les coordonnées de texture du cube.

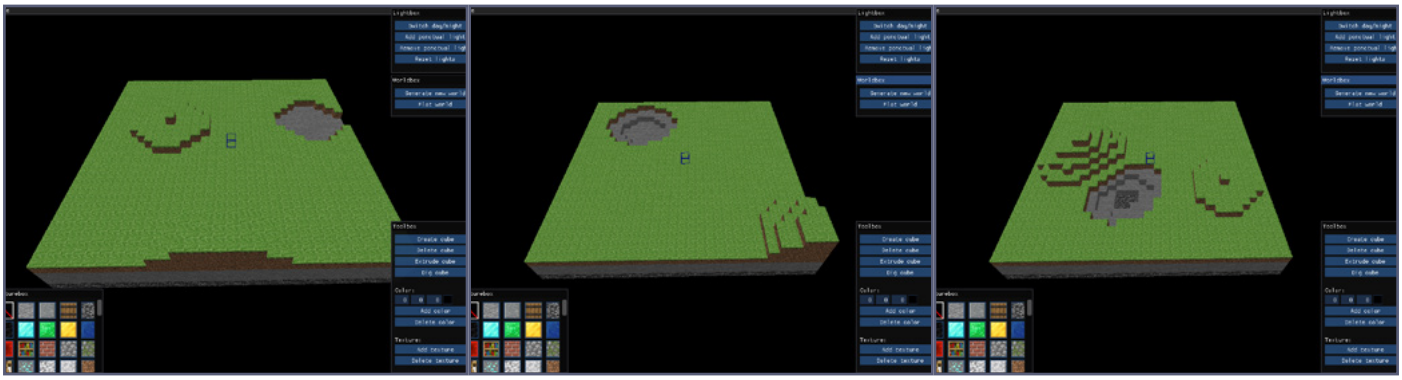
Ainsi, grâce à ces deux points, nous avons pu faire un **vecteur de toutes les références des textures** que nous voulions utiliser, avec aussi leurs noms et leurs références de textures pour l'interface (une simple texture 2D à afficher en bouton radio). **Ce tableau se remplit automatiquement** en allant chercher les 80 textures dans le dossier /assets/textures/cube_textures, et ce grâce à une boucle et des fonctions du **FileSystem de la librairie Boost**. Tout ceci a été codé dans Texture.cpp.

Pour finir, il a fallu modifier Interface.cpp pour pouvoir **visualiser et choisir les textures grâce à une fenêtre de boutons radio** (Texturebox). Comme pour les couleurs, l'interface exécute donc des fonctions dans Scene.cpp avec les caractéristiques choisies par l'utilisateur. Ces fonctions peuvent agir grâce à l'id d'une texture (position dans le vecteur des textures), mais aussi grâce au nom d'une texture. Nous avions hésité à faire une map au départ, mais vu que les shaders utilisent seulement un numéro pour identifier les textures, il était finalement plus judicieux d'utiliser un vecteur.

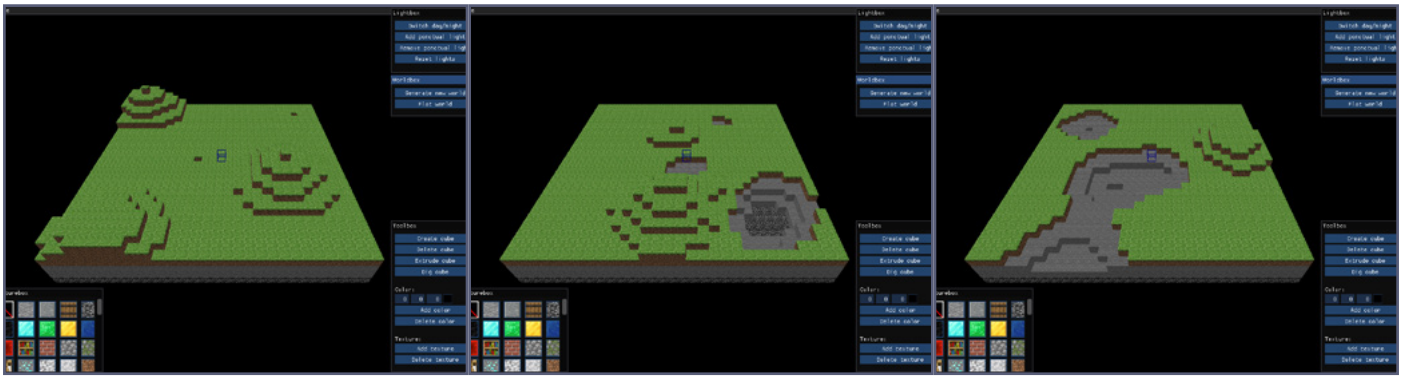




/Radial basic functions

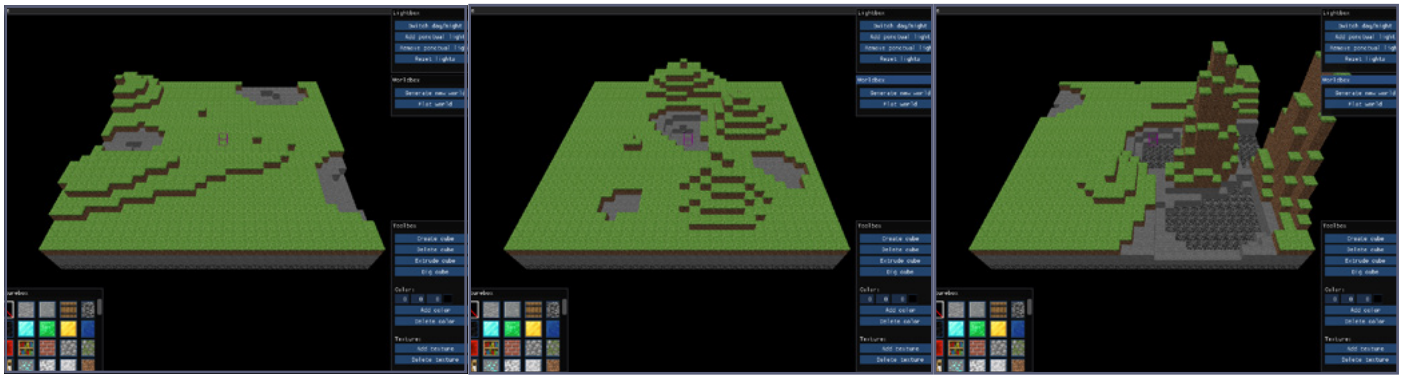
/Générations avec les RBF



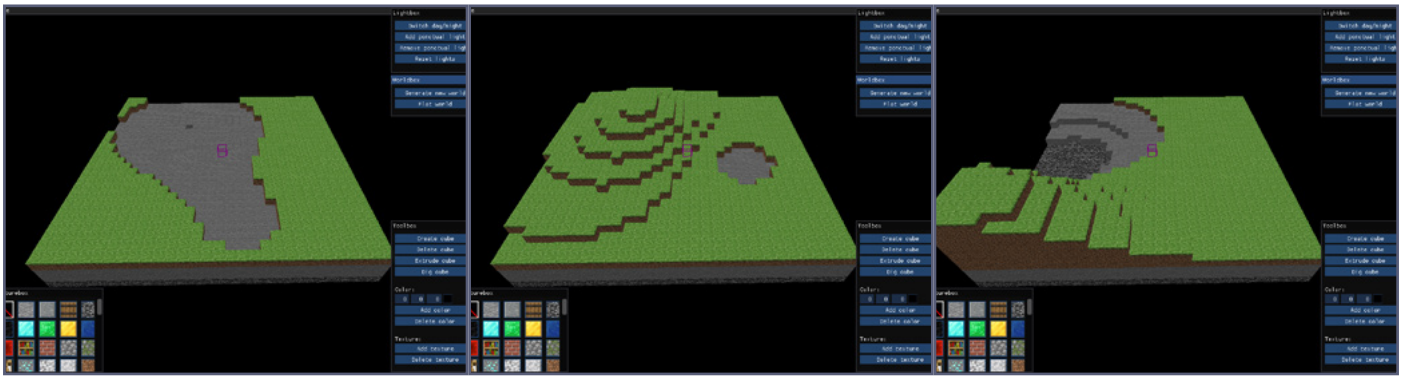
Fonction gaussienne : 3 points



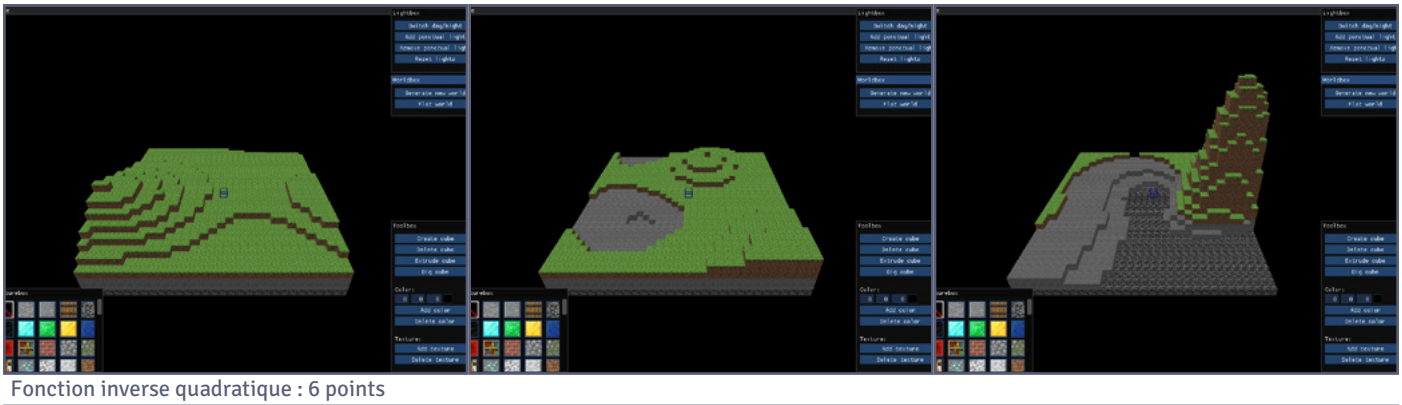
Fonction gaussienne : 6 points



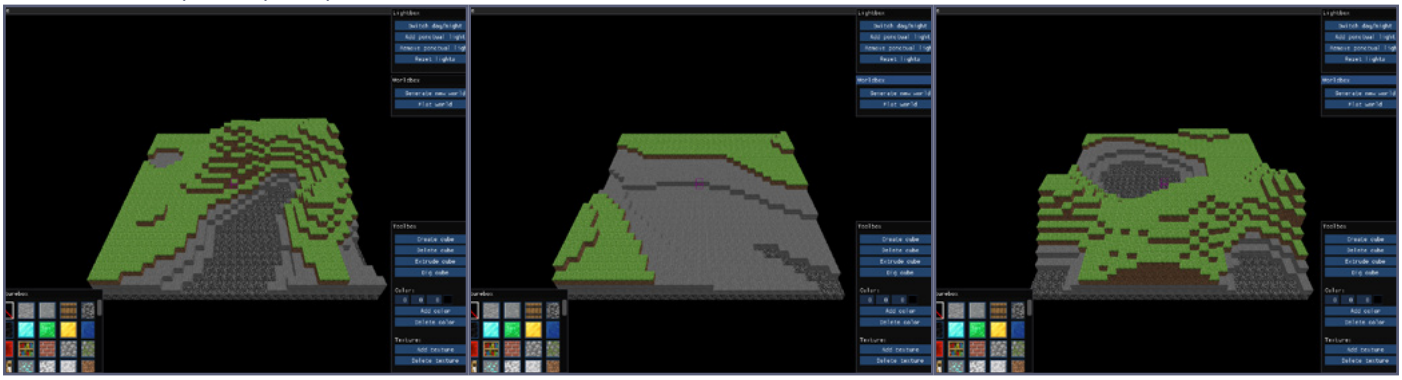
Fonction gaussienne : 12 points



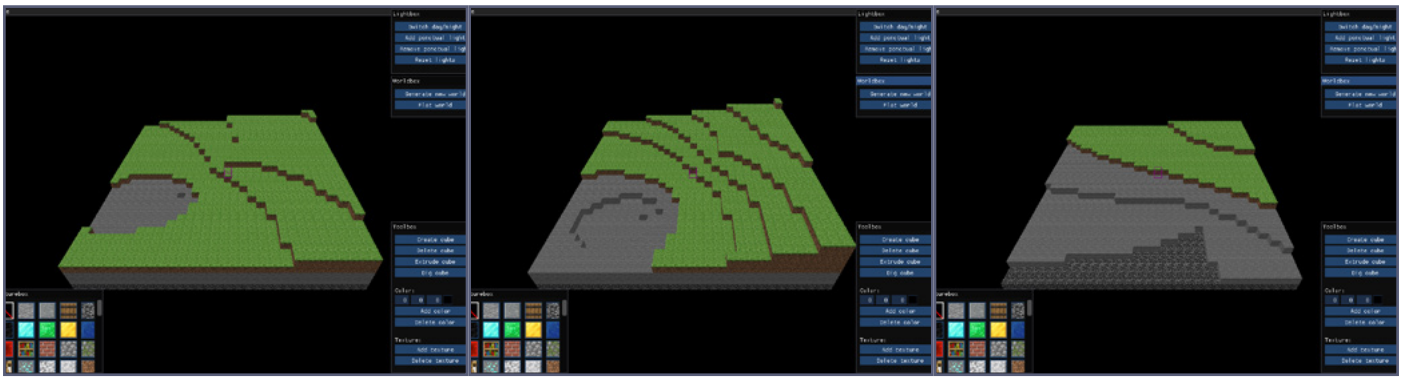
Fonction inverse quadratique : 3 points



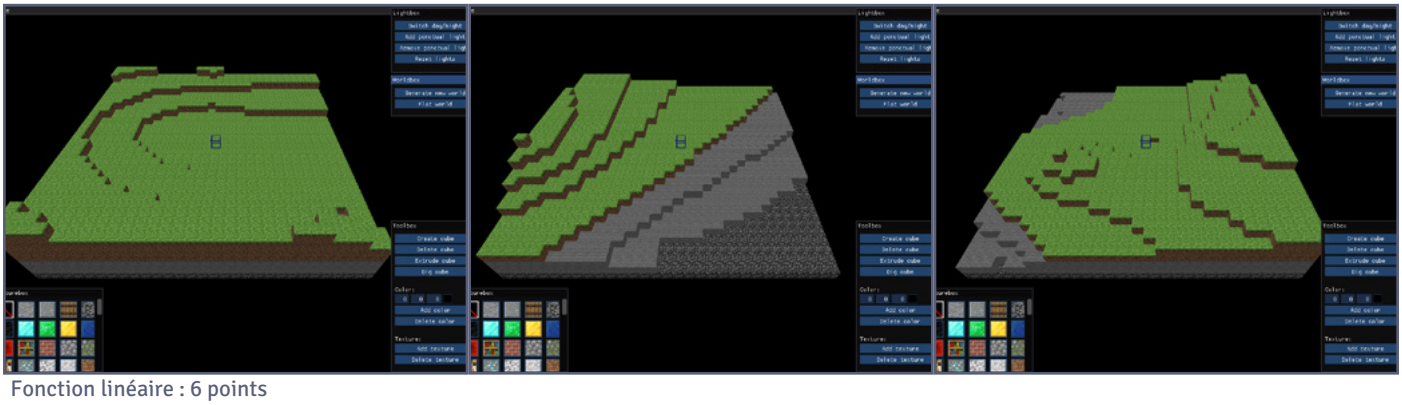
Fonction inverse quadratique : 6 points



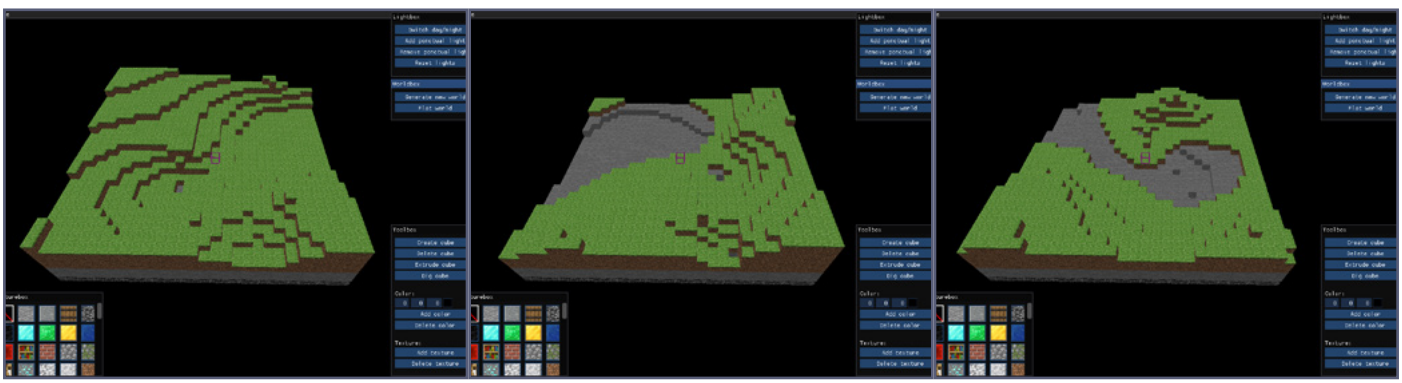
Fonction inverse quadratique : 12 points



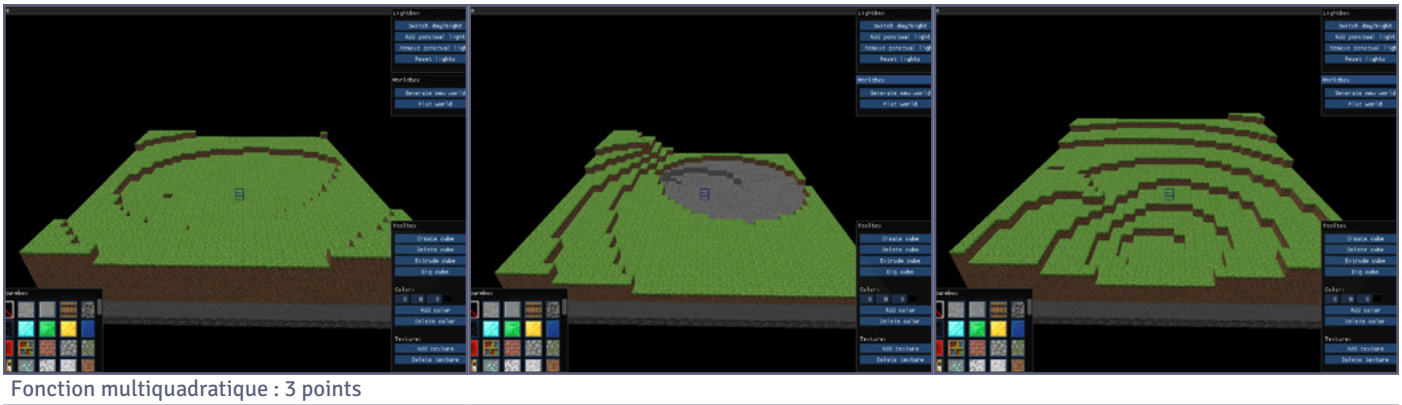
Fonction linéaire : 3 points



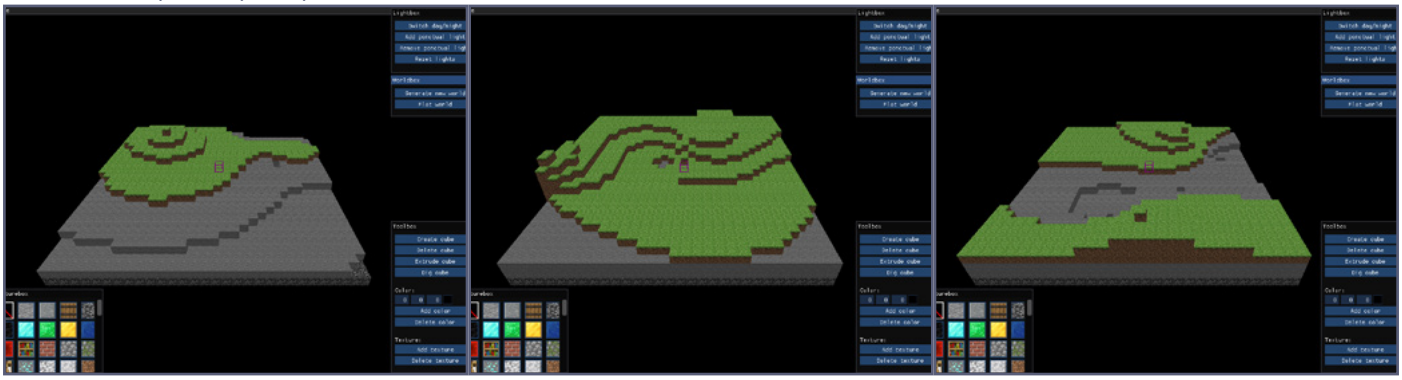
Fonction linéaire : 6 points



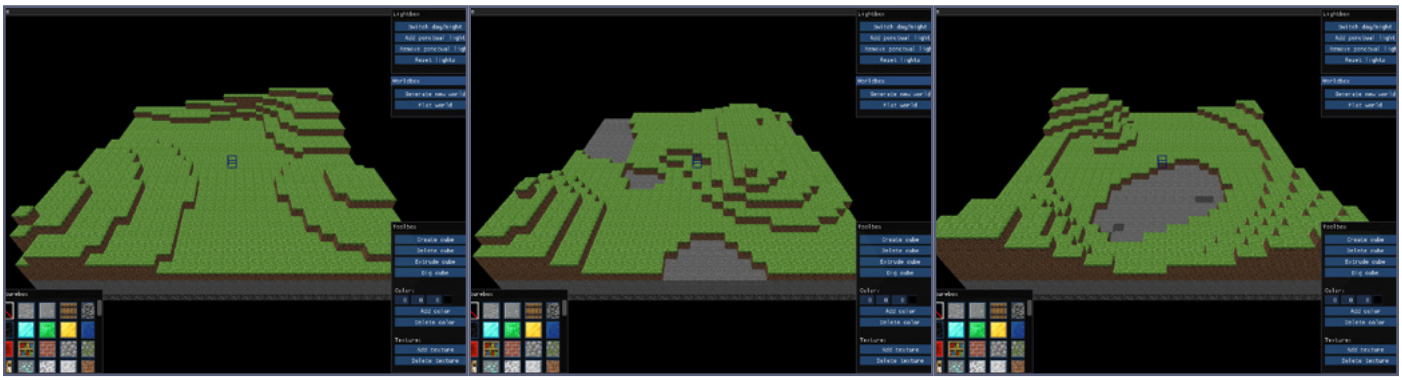
Fonction linéaire : 12 points



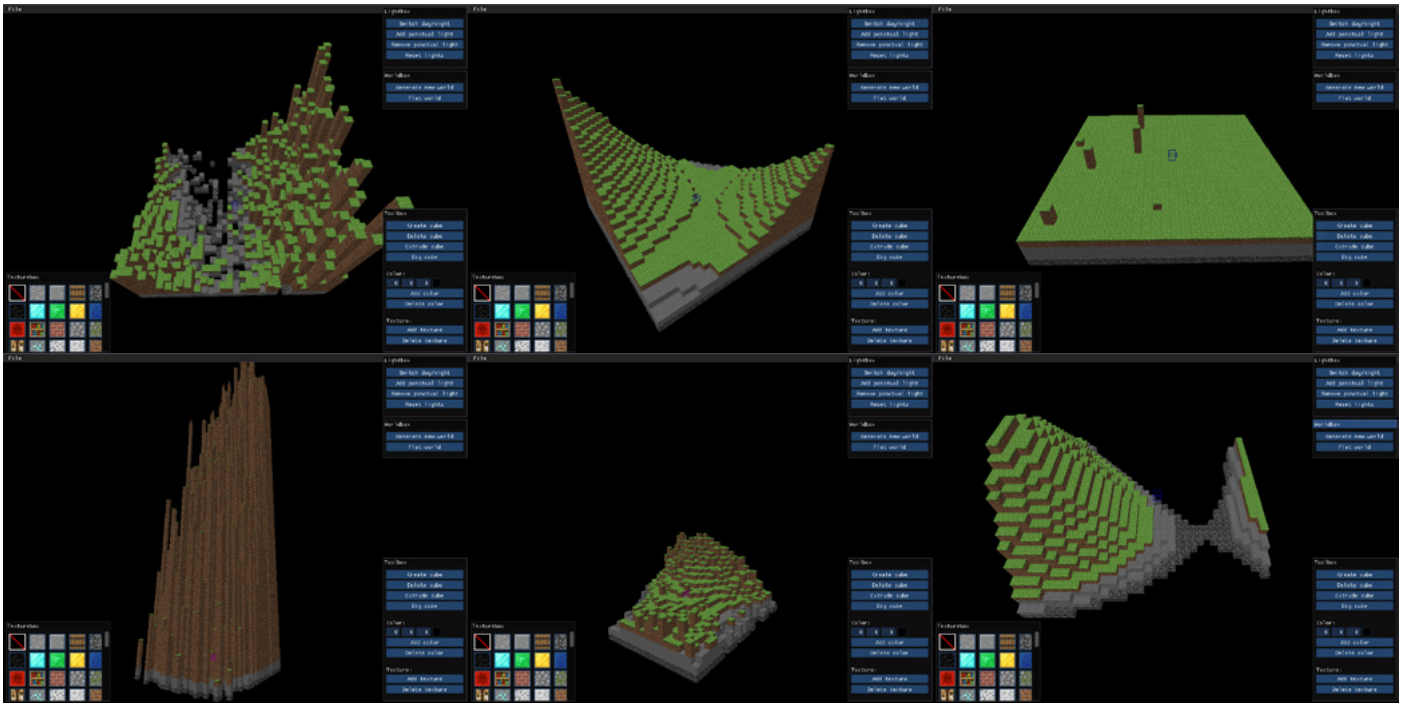
Fonction multiquadratique : 3 points



Fonction multiquadratique : 6 points



Fonction multiquadratique : 12 points



Expérimentations avec les RBF :
inverse quadratique epsilon 0,000 1 / inverse quadratique epsilon 0,001 / inverse quadratique epsilon 50
multiquadratique epsilon 0,000 1 / multiquadratique epsilon 0,000 1 (V2) / gausienne epsilon 0,000 1

/Analyse du rendu des fonctions

Linéaire :

Bien pour faire des plaines, petites collines et petits lacs. Nous avons choisi des poids de départ assez faibles (entre -3 et 4). Avec des poids plus grands (en valeur absolue), nous pouvons générer des **cuvettes**.

Multiquadratique :

Génère des terrains ressemblant parfois à des théâtres antiques, parfois à des petites butes. De fois, c'est assez plat et avec des « marches » : cela peut faire penser à un bord de mer ou de rivière (epsilon égal à 5). Mettre un epsilon grand n'est pas très intéressant. Avec un epsilon petit, des piliers se forment (exemple : 0,000 1) : on peut imaginer des immeubles ou bien parfois des petites bâtisses.

Inverse quadratique :

C'est assez imprévisible. On peut avoir des terrains assez différents : parfois assez plats, parfois des petites montagnes, parfois des trous. Nous avons choisi un epsilon égal à 0,1 pour ne pas avoir de valeurs trop grandes (en valeur absolue). Avec un epsilon grand (50 par exemple), les points de contrôle ressortent, mais le reste est plat : ce n'est pas très intéressant. Avec un epsilon plus petit (0,001), nous pouvons générer des montagnes. Cependant, les valeurs peuvent exploser et cela prend alors du temps d'afficher la nouvelle scène.

Gaussienne :

Nous obtenons souvent des monticules ou des trous isolés sur la plaine. Sans cette configuration plate au départ, nous aurions surement des îlots. Nous avons choisi un epsilon égal à 0,04 pour garder quelque chose d'à peu près constant. Avec un epsilon grand, nous observons la même chose qu'avec la fonction inverse quadratique. Avec un epsilon petit, nous pouvons former des montagnes et des gouffres. Mais, les valeurs peuvent là aussi exploser : c'est donc assez imprévisible et pas très adapté à notre programme au vu du temps d'affichage nécessaire.

Remarques générales :

Plus il y a de points de contrôle, plus le terrain aura de relief. Cela donne donc des scènes intéressantes. Mais, comme notre scène de départ n'est pas très grande, **nous avons mis à 6 le nombre de points de contrôle par défaut** dans le programme.

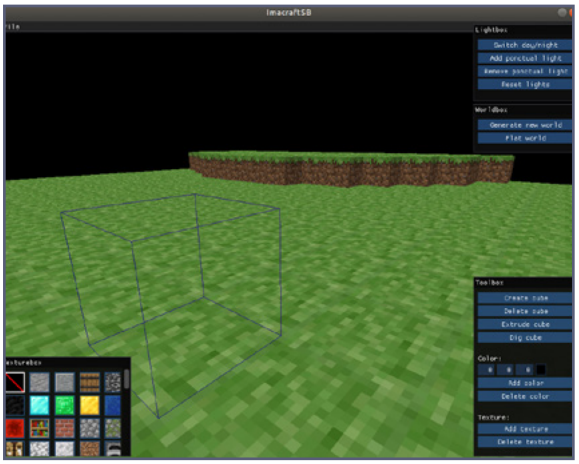
Comme fonction par défaut, nous avons choisi **la fonction quadratique avec un epsilon égal à 5** car nous trouvions que c'était celle qui donnait les résultats les plus différents les uns des autres tout en restant assez « stable » au niveau des altitudes générées.

Nous remarquons qu'avec les fonctions multiquadratique, inverse quadratique et gaussienne, les **poids interpolés peuvent être très grands, alors que nous avons sur les points de contrôle des poids assez faibles** en comparaison (toujours entre -3 et 4). C'est probablement dû au phénomène de Runge.

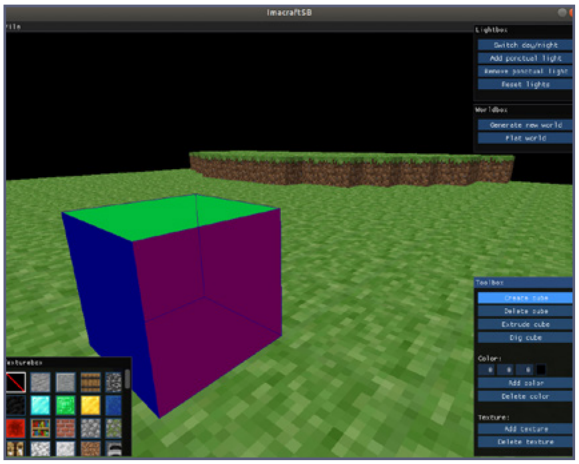


/Résultats

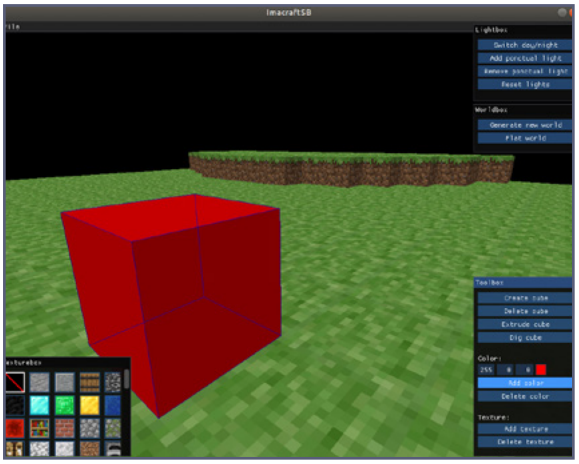
/Toolbox et Texturebox



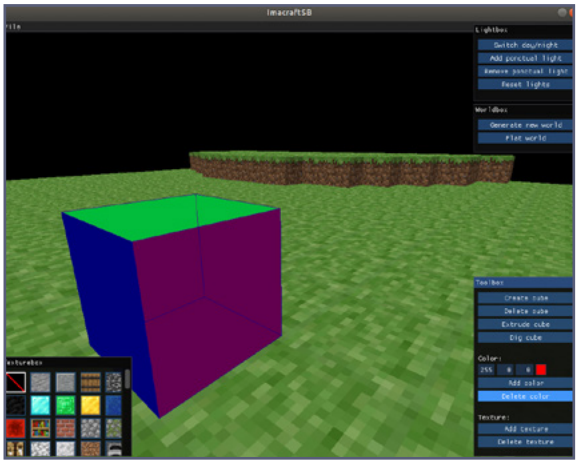
Curseur vide



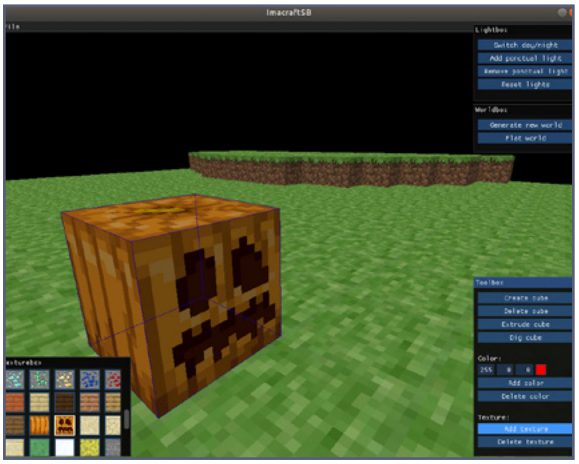
Créer un cube sans caracté-ristiques



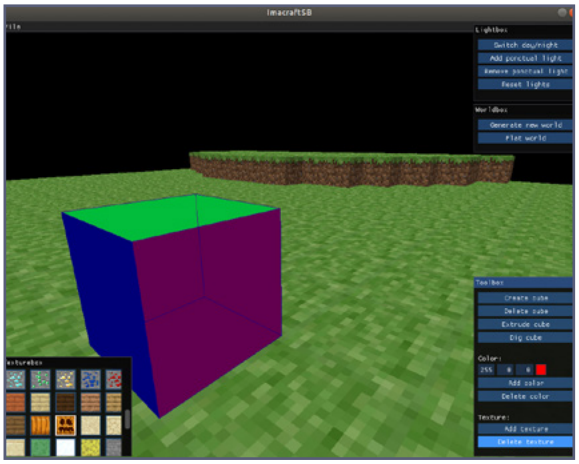
Ajouter une couleur



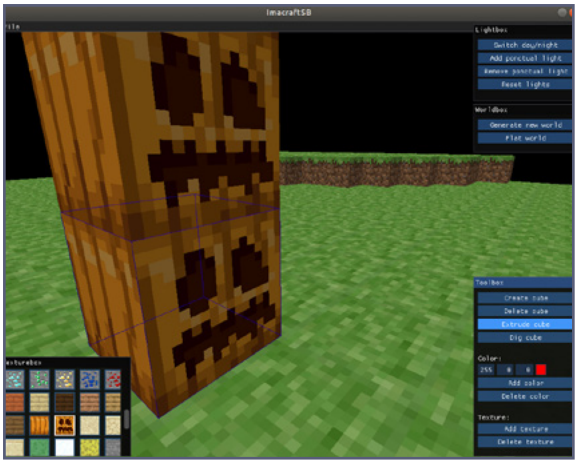
Supprimer la couleur



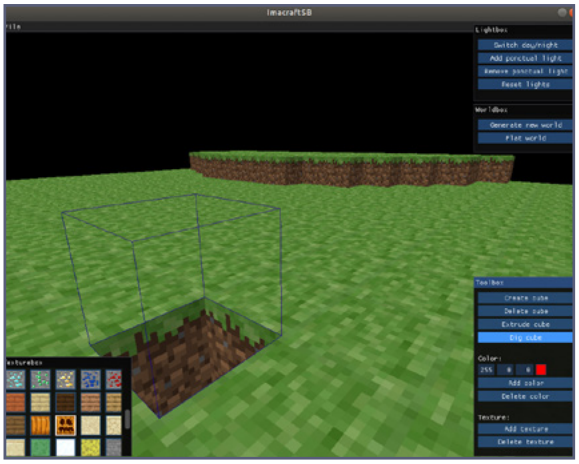
Ajouter une texture



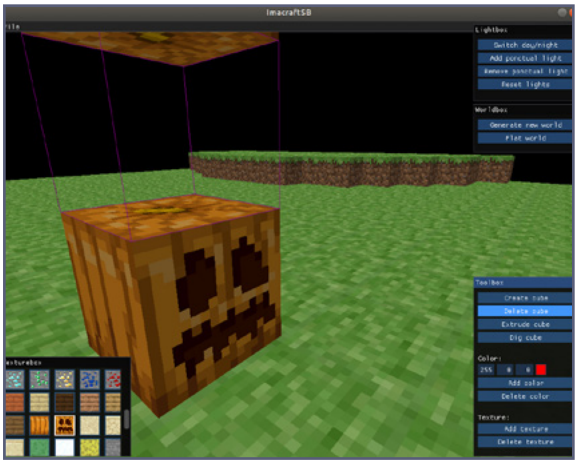
Supprimer la texture



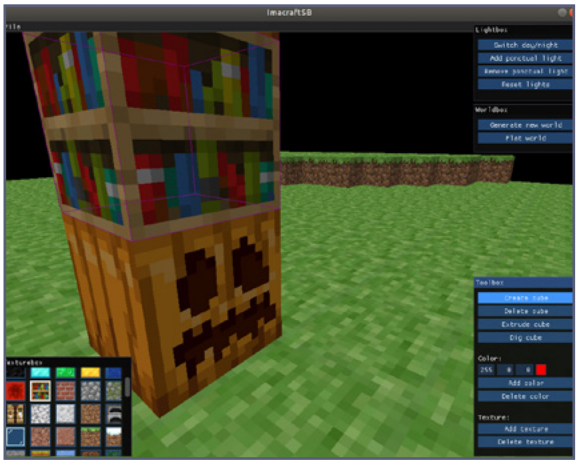
Extruder le cube (extrude)



Creuser le cube (dig)

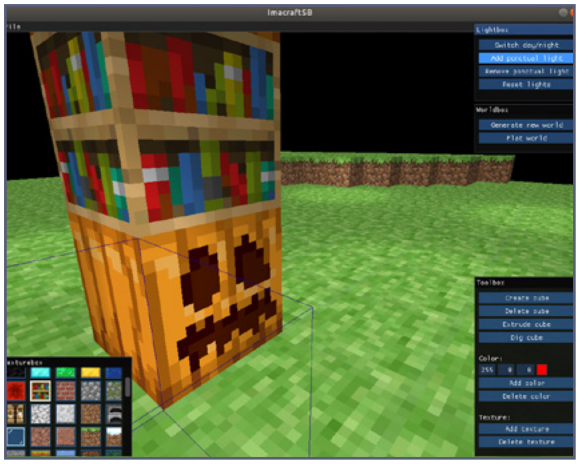


Supprimer un cube

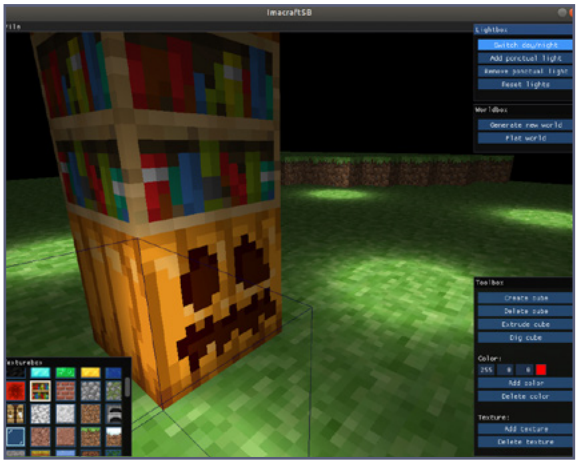


Créer un cube avec une texture

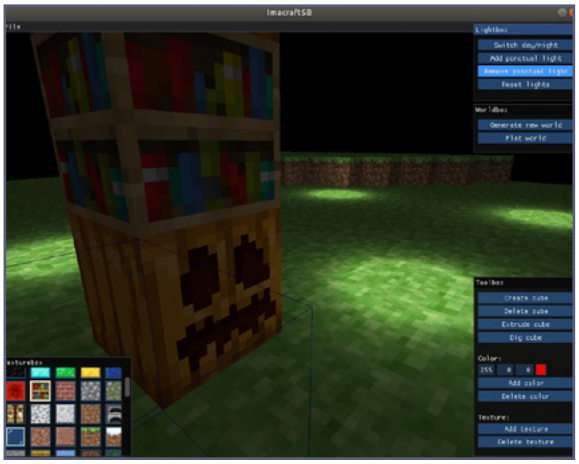
/Lightbox et Worldbox



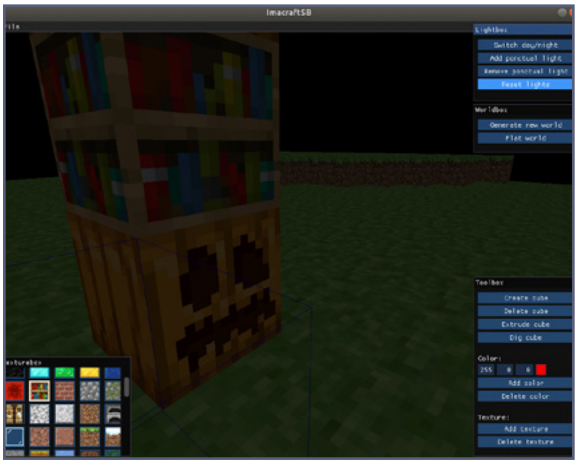
Créer une lumière ponctuelle



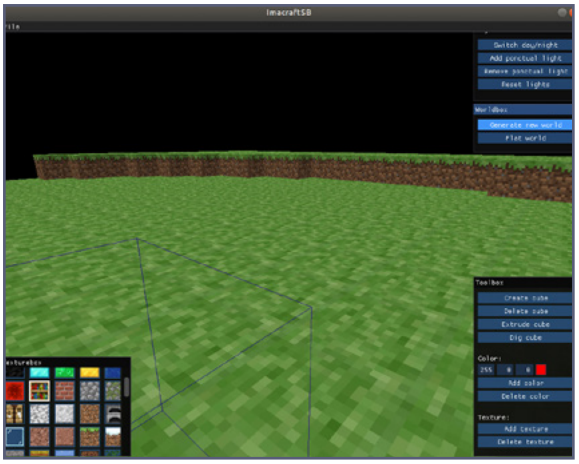
Passer en mode « nuit »



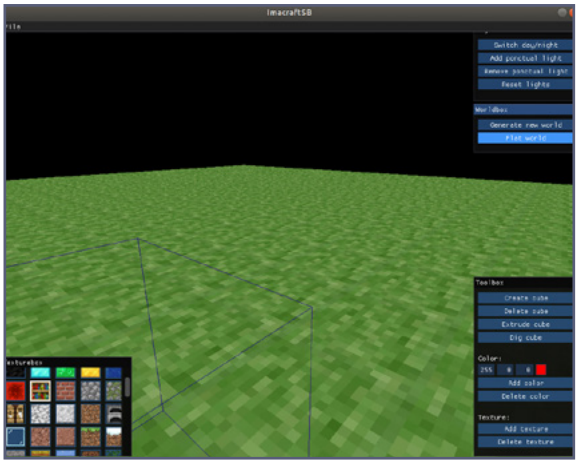
Supprimer une lumière ponctuelle



Réinitia-liser les lumières

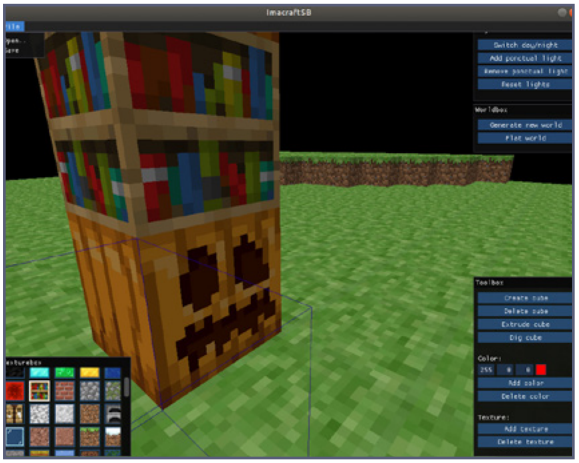


Générer un nouveau monde aléatoire

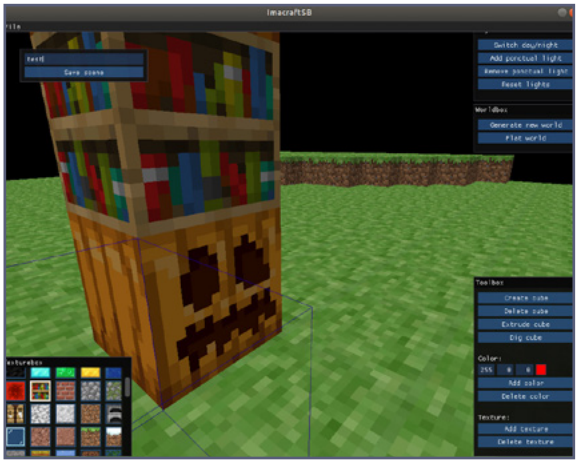


Générer un nouveau monde plat

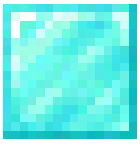
/Barre de menu



Ouvrir le menu « Fichier » (barre de menu)



Sauvegarder une scène



/Conclusion

/Difficultés rencontrées

Nous avons rencontré **quelques difficultés au cours de ce projet, mais elles ont toutes fini par être réglées ou contournées**. Nous en avons déjà évoqué quelques-unes dans les parties précédentes, mais en voici d'autres plus générales.

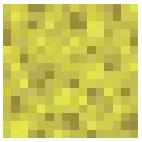
Tout d'abord, il a été **compliqué de définir une architecture cohérente pour le projet**. En effet, nous voulions que notre projet soit propre et facilement compréhensible, en respectant les quatre premiers patterns de GRASP (expert — faible couplage — forte cohésion - création). Il a donc parfois été compliqué de faire cela, et ce **tout en utilisant OpenGL et la SDL qui forcent à travailler les fonctions d'une façon bien particulière**.

Et, malgré les nombreux exemples disponibles sur internet, nous avons aussi eu **un peu de mal à utiliser les librairies** : liens avec le CMake, entrée d'un texte par l'utilisateur et boutons radio avec des images (imgui), utilisation des fonctions du système de fichiers (Boost), etc.

Et finalement, **certains calculs et codes ont été fastidieux à créer ou appliquer** : les lumières, la génération procédurale, les fonctions de la toolbox, etc. **Optimiser tout cela** également : le déplacement peut être vite très lent s'il y a beaucoup de lumières.

/Améliorations

Nous avons en tête quelques **améliorations pour ce projet** : alléger et optimiser les lumières (avec un g-buffer par exemple), améliorer le curseur en faisant une sélection à la souris (comme dans Minecraft), faire hériter le curseur du cube, permettre la sauvegarde et le chargement des scènes en spécifiant l'endroit et en voyant les autres fichiers (une sorte d'explorateur de fichiers), faire une fenêtre caméra pour choisir un angle de caméra ou changer de type de caméra, améliorer la transparence des cubes (avec un z-buffer par exemple), ajouter des fonctionnalités à la Worldbox (choix de la fonction radiale, choix des paramètres, etc.) et optimiser/améliorer l'architecture globale et les fonctions pour fluidifier le programme.

/Nos retours

Baptiste

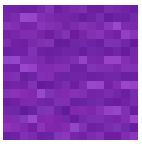
J'ai beaucoup aimé que le projet soit très « visuel » et ainsi pouvoir voir ce que je codais prendre forme sur l'écran. De plus, je me suis aussi beaucoup amusé avec la librairie imgui en changeant le style, positionnant et redimensionnant les éléments ; un peu à la manière de ce qu'on peut faire en web.

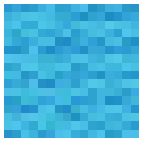
Qui plus est, je pense avoir beaucoup appris en programmation C++ et en synthèse d'images en appliquant ce qu'on avait vu en cours et en cherchant à faire les choses que j'avais en tête.

Solène

Pouvoir appliquer de manière plus concrète ce que l'on a vu pendant ce semestre était assez plaisant, je pense que cela m'a permis de mieux comprendre ce à quoi tout ce qu'on a appris peut servir dans un « vrai » projet. Même si j'ai pu me perdre un peu dans la librairie imgui, l'utiliser était — pour moi — assez instructif, mais surtout agréable lorsque cela marchait.

Tout de même, je pense que la partie sur laquelle j'ai préféré travailler était les « radial basis functions ». Cela est très certainement lié au fait qu'elles étaient utilisées pour de la génération de terrain : voir toutes mes petites fonctions réussir à afficher un terrain différent à chaque fois était très satisfaisant. Si cela avait été pour une courbe sur un plan 2D, j'aurai probablement trouvé ça moins amusant.





/Sources

Cours de « Programmation objet », « Mathématiques pour l'informatique » et « Synthèse d'images » 2019-2020, en IMAC 2 — semestre 3 : Vincent Nozick, Venceslas Biri et Céline Noël.

Auteurs Cppreference.com. C++ reference : cppreference.com [en ligne]. Disponible sur : https://en.cppreference.com/w/.

Auteurs Développez .com. Cours de C/C++ : Developpez.com [en ligne]. 22/11/2012 (modif.). Disponible sur : https://cpp.developpez.com/cours/cpp/.

Auteurs Eigen. Eigen : Quick reference guide : Eigen [en ligne]. Disponible sur : https:/eigen.tuxfamily.org/dox/group__QuickRefPage.html.

Auteurs OpenGL-tutorial. Page principale : OpenGL-tutorial [en ligne]. 07/06/2017 (modifs.). Disponible sur : http://www.opengl-tutorial.org/fr/.

Auteurs OpenGL Wiki. OpenGL Wiki : OpenGL Wiki [en ligne]. Disponible sur : http://khronos.org/opengl/wiki/.

CORNUT, Aymar. ocornut / imgui : GitHub [en ligne]. 25/12/2019 (modif.). Disponible sur : https://github.com/ocornut/imgui.

CHRISTEN, Martin. Clockworkcoders : Texturing : OpenGL Software Development Kit (SDK) [en ligne]. 2007. Disponible sur : https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/texturing.php.

DE VRIES, Joey. Learn OpenGL : Learn OpenGL [en ligne]. 2014. Disponible sur : https://learnopengl.com/.

GOUJEON, Florian. Initiation à CMake : Developpez.com [en ligne]. 06/01/2009. Disponible sur : https://florian-goujeon.developpez.com/cours/cmake/initiation/.

JAWORSKI, Michael. Using input text flags : pyimgui [en ligne]. 2016. Disponible sur : https://pyimgui.readthedocs.io/en/latest/guide/inputtext-flags.html.

NEBRA, Mathieu et SCHALLER, Matthieu. Programmez avec le langage C++ : OpenClassrooms [en ligne]. 30/09/2019 (modif.). Disponible sur : https://openclassrooms.com/fr/courses/1894236-programmez-avec-le-langage-c?status=published.

NOËL, Laurent. OpenGL3+ : Université Paris-Est Marne-la-Vallée [en ligne]. Disponible sur : http://igm.univ-mlv.fr/~lnoel/index.php?section=teaching&teaching=opengl&teaching_section=intro.

NOZICK, Vincent. Radial Basis Functions : Université Paris-Est Marne-la-Vallée [en ligne]. Disponible sur : http://www-igm.univ-mlv.fr/~vnozick/divers/rbf.pdf.

POUET_FOREVER et KARNAJ (Anonymes). Utiliser la SDL en langage C : Zeste de savoir [en ligne]. 27/12/2018 (modif.). Disponible sur : https://zestedesavoir.com/tutoriels/1014/utiliser-la-sdl-en-langage-c/.

Lien github du projet : https://github.com/SolHaine/ProjetProg_ImacraftSB