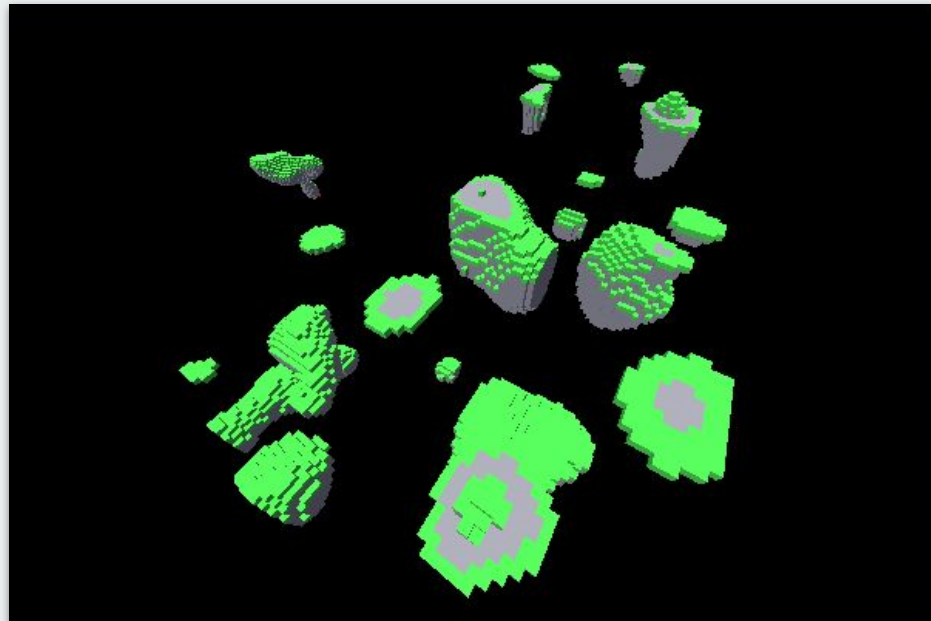


Comment faire un moteur Voxel 3D ? (comme Minecraft)

Daniel Braikeh





Définition et exigences

Un voxel est un pixel 3D. Le voxel est à la 3D ce qu'un pixel est à la 2D.

Un moteur voxel permet d'afficher des mondes composées de voxel et de les éditer en temps réel. Les mondes sont souvent destinés à être très grands, voire infinis.

Minecraft a fait le choix d'être simplifié en limitant la hauteur maximale et minimale des voxels. C'est à dire que, par exemple, l'on peut construire entre une altitude de -128 mètres et une altitude de 128 mètres, avec le niveau du sol et de la mer à 0 mètres.

Dans mon moteur, je fais le choix de ne pas me limiter en hauteur.

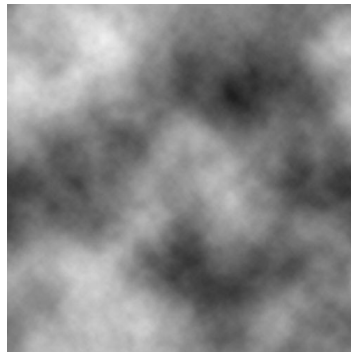
Comment générer un monde


Nous souhaitons pouvoir générer un monde très grand de manière automatisée, et qui soit cohérent et agréable au joueur. C'est à dire qu'un simple `random()` sera insuffisant.

Nous utilisons donc la technique basée sur le bruit de Perlin ([Perlin noise](#)) qui permet de générer une information continue et cohérente (voir l'image). Il suffit d'itérer sur chaque pixel de l'image pour commencer à générer notre monde.

A partir de ces informations, nous choisissons les seuils pour définir les voxels. Par exemple, avec une valeur entre 0.0 et 1.0,

- $[0.0, 0.2]$ = vide
- $]0.2, 0.8]$ = roche
- $]0.8, 1.0]$ = herbe



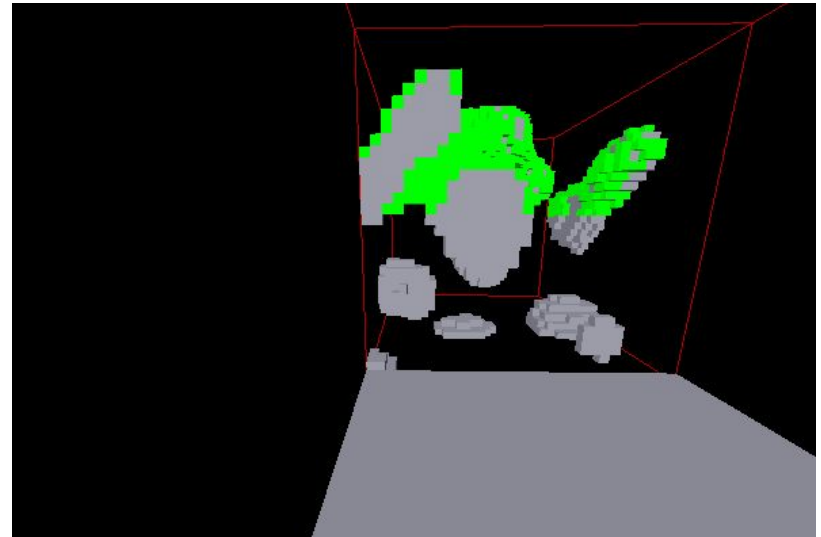


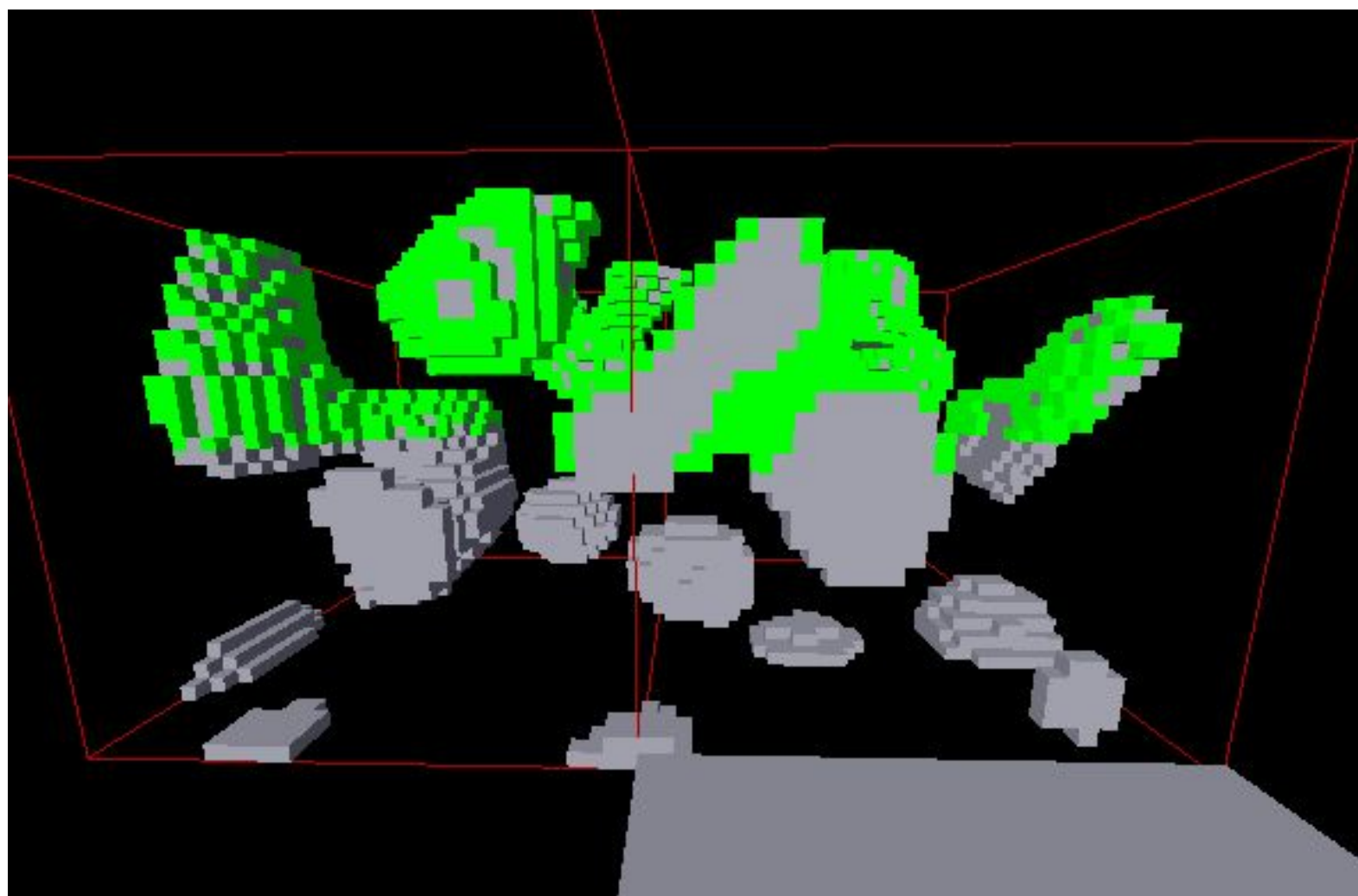
Nous utilisons donc un bruit de Perlin 3D et cela nous permet d'obtenir des îles flottantes.

Sur l'exemple, nous itérons sur Perlin de 0 à 32.

Et comme le bruit de Perlin est une simple courbe mathématique infinie, il n'y a rien à sauvegarder pour régénérer cette partie du monde.
(il suffit seulement de sauvegarder les facteurs de notre bruit)

Pour générer une suite cohérente du monde, il suffit de continuer à explorer le bruit de Perlin.







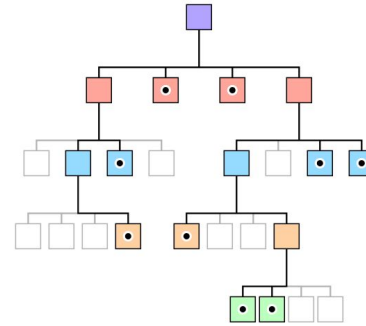
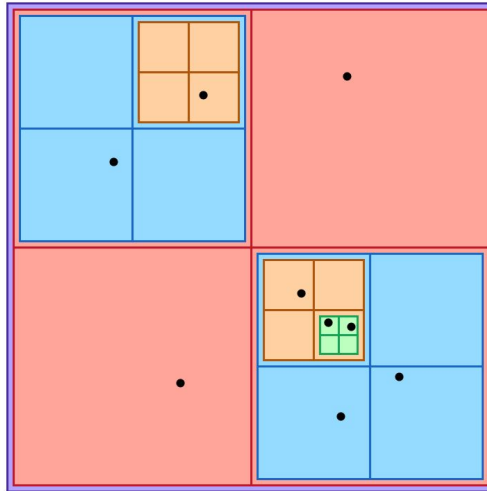
Comment mettre le monde en mémoire ?

On imagine bien qu'il est impossible de générer le monde en entier et de le garder entièrement en mémoire dans un tableau. Il faut donc obligatoirement générer et mémoriser seulement ce qui est nécessaire au joueur.

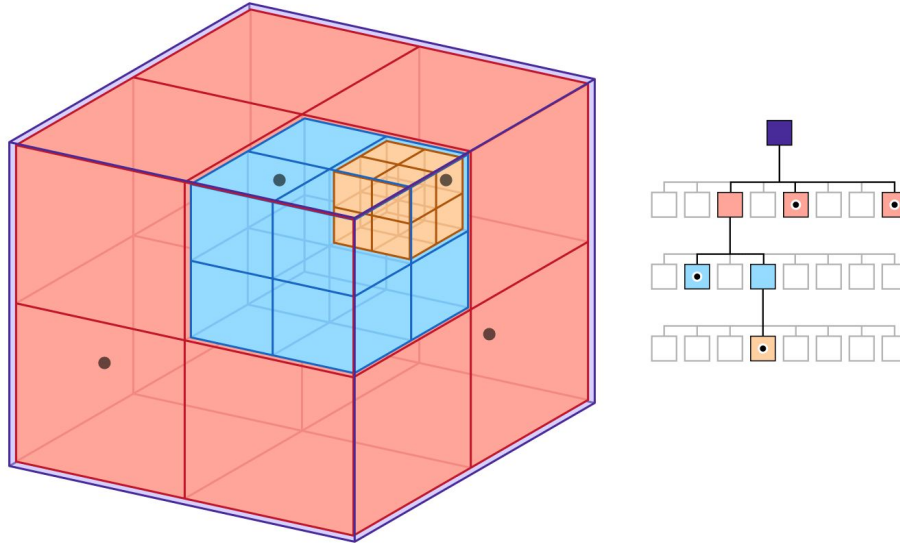
L'hypothèse primordiale est que l'on doit gérer uniquement le monde qui se trouve autour du joueur (dans un certain rayon, ou sphère puisque l'on est en 3D). Ainsi seulement ce qui est "visible" sera généré et mémorisé. Et lorsque le joueur s'éloigne, le monde devenu "éloigné" sera supprimé de la mémoire.

Une autre hypothèse importante est que la vitesse de déplacement du joueur est limitée, ce qui laisse le temps au moteur de générer les parties du monde devenues visibles (on peut aussi tirer avantage qu'elles sont les plus éloignées du joueur).

Une première technique est d'utiliser un arbre spatial qui nous permet d'éviter de mémoriser du vide. Pour la 2D, il existe les quadtrees qui permettent de partitionner l'espace 2D.



En 3D, l'équivalent est un **Octree** (arbre à huit branches). Ainsi, les cubes oranges seraient nos voxels.





Optimisation : le Chunk

L'accès en lecture / écriture aux voxels dans un octree peut être pénalisante.

En effet, pour accéder à un voxel, il faut parcourir tout l'arbre, ce qui se fait en $O(h)$ avec h la hauteur de l'arbre. Et si l'on souhaite accéder au voxel voisin, il faut faire la même opération.

Pour optimiser ces accès, nous allons réunir un petit ensemble de voxels dans un tableau en mémoire. Ainsi, localement autour du joueur, les accès seront directs en $O(1)$.

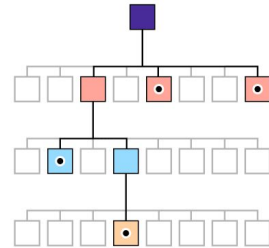
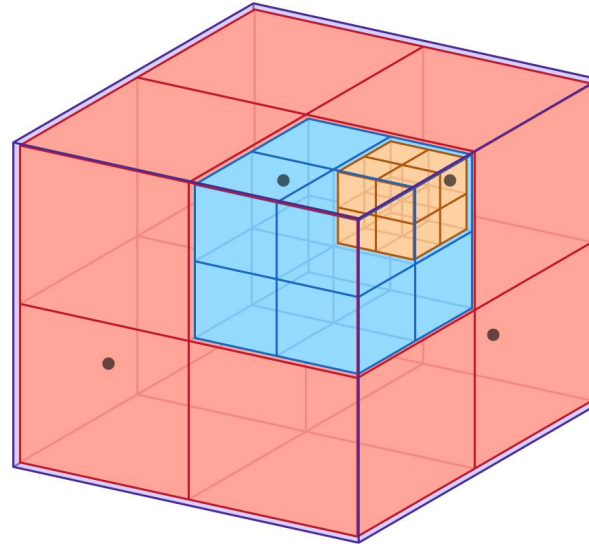
Nous réunissons donc $32 * 32 * 32$ voxels en un ensemble appelé **Chunk**.

(32 étant une puissance de 2, cela permettra d'accélérer les calculs grâce aux décalages de bits.)



Ainsi, sur cette image, notre Chunk sera le cube orange contenant le point noir. On voit qu'il se trouve au plus profond de l'arbre.

Le reste étant du vide, on a économisé tous ces chunks en mémoire.





Comment afficher en 3D ?

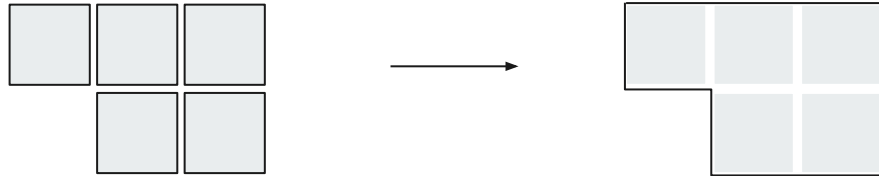
Une technique naïve d'affichage serait de parcourir tous les voxels et d'envoyer le cube à afficher à la carte graphique. Le problème étant le nombre énorme de voxels et que cette opération est en $O(n^3)$! Il nous faut trouver une autre technique.

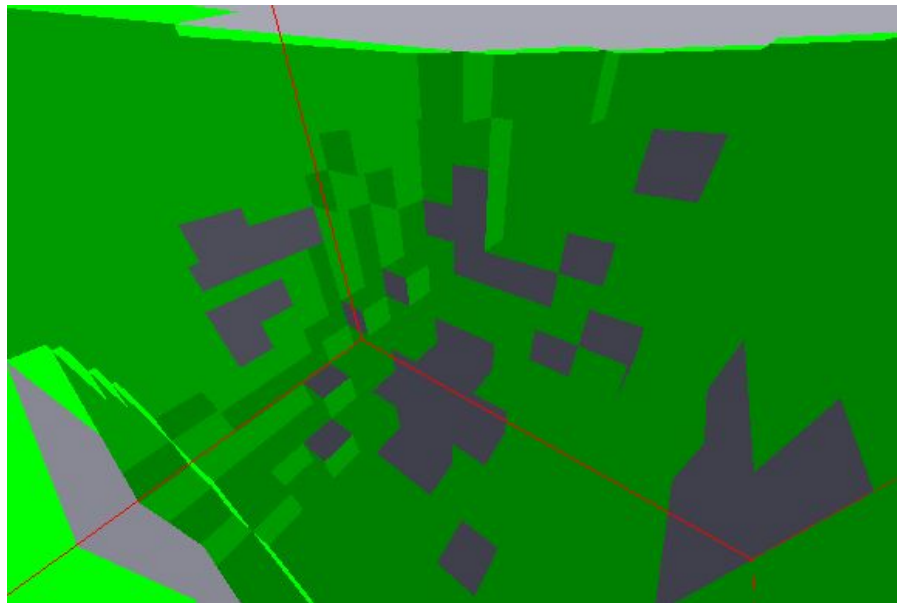
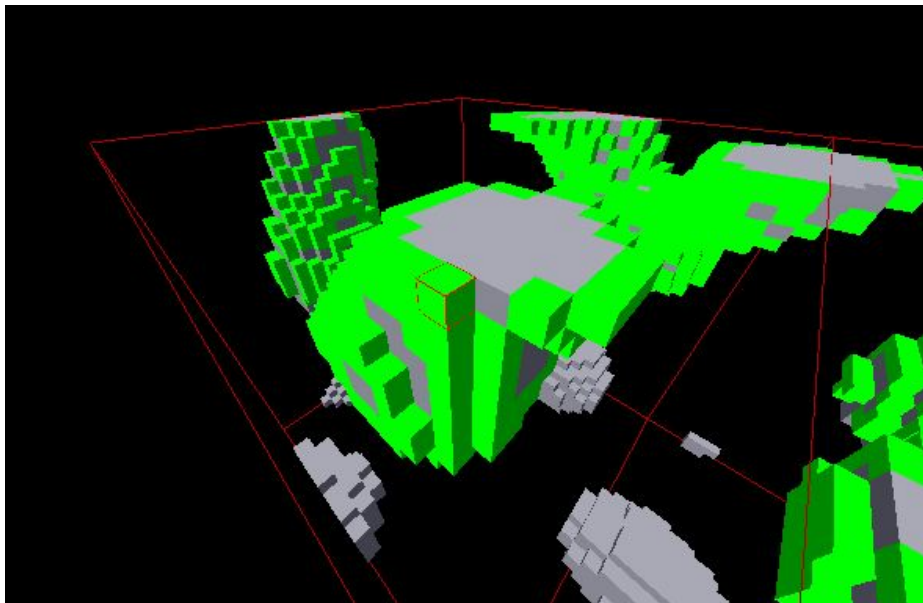
Deux hypothèses nous permettent d'optimiser cela :

1. Le joueur ne voit pas ce qu'il y a à l'intérieur des îles. Il n'y a donc pas besoin de rendre les cubes à l'intérieur de la "masse" de voxels.
2. Les voxels sont rarement modifiés. Le joueur va souvent se concentrer sur un voxel en particulier. Les voxels sont rendus 60 fois par seconde tandis que le joueur modifiera un voxel une fois par seconde.



Nous pouvons alors construire une enveloppe des voxels pour réduire le nombre de faces à afficher en 3D.





L'intérieur d'une île



Une fois cette enveloppe calculée, nous pouvons la sauvegarder dans la mémoire de la carte graphique sous forme d'objet identifiable.

Ainsi, lors du rendu d'un Chunk (60 fois par seconde), au lieu d'envoyer toutes les faces, nous donnons simplement l'ID de l'objet à dessiner.

Lorsqu'un voxel de ce Chunk sera modifié, il suffira de recalculer la nouvelle enveloppe.



Repousser encore les limites

L'octree, par sa construction, nécessite de connaître la taille maximale de ce qu'il aura à stocker. Cela limite donc notre monde à une certaine taille (même si elle est quand même de $32^3 \cdot 4096$ voxels ou 131 km)

Une autre problématique est la performance. Comme nous l'avons vu, un accès dans un octree est en $O(h)$. Donc pour accéder aux Chunks voisins autour du joueur, nous aurons plusieurs opérations en $O(h)$.

Afin de repousser cette limite, nous allons utiliser la technique de la [pagination virtuelle](#). C'est la même technique utilisée par l'Operating System pour gérer la mémoire RAM qui est virtuellement infinie pour chaque processus.

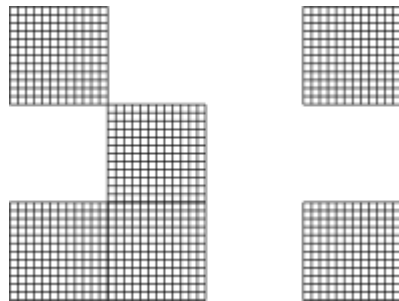


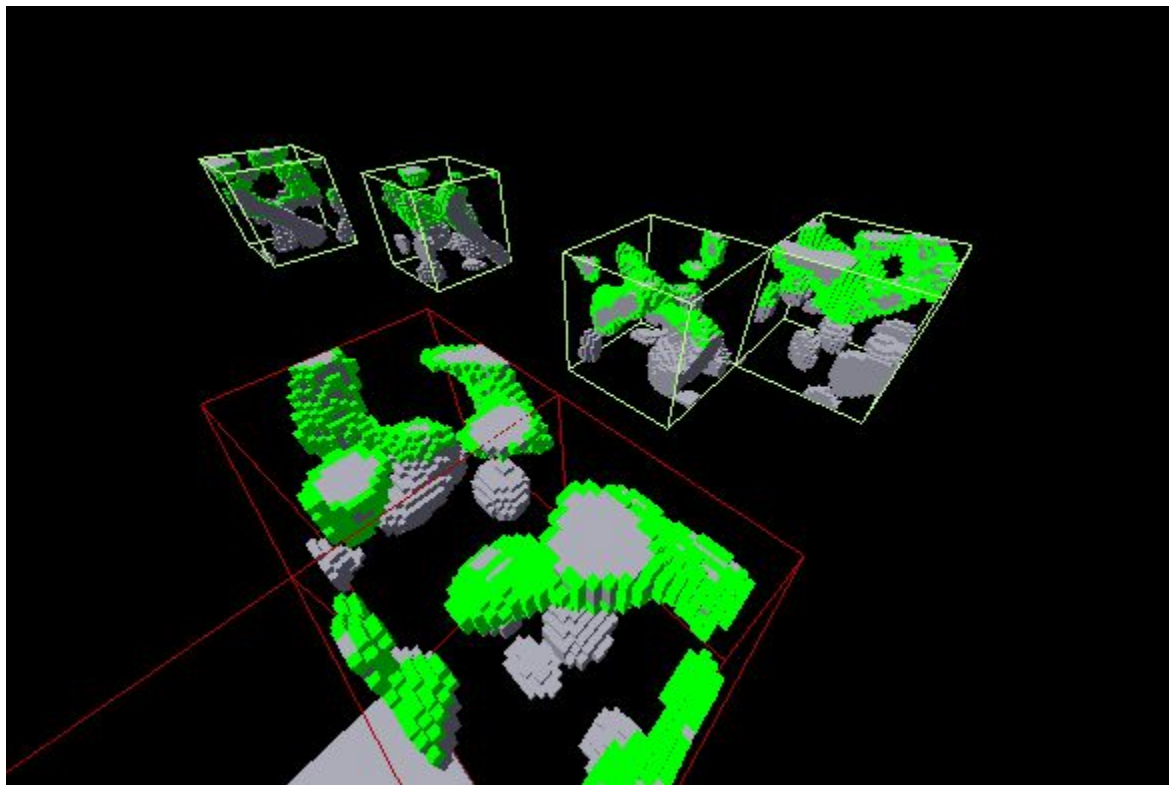
Virtual Pagination

L'idée est de découper notre monde en une grille 3D infinie et régulière. Seules les cases qui existent réellement sont en mémoire. Les autres existent virtuellement et seront chargées à la demande.

Nous avons donc une simple **HashMap** qui a pour clé une position 3D et pour valeur un Chunk. L'avantage est qu'un accès dans une HashMap est en $O(1)$.

Nous avons ainsi, pour n'importe quel voxel existant et en mémoire, un accès en $O(1)$. C'est comme si nous avions effectivement un immense tableau en mémoire.





Nous avons donc un moteur voxel 3D performant.

*Il reste bien d'autres sujets à voir, comme l'ajout ou la suppression d'un voxel par le joueur,
la persistance du monde, ...*