



Ft_vox

Pimp my blocks

rcargou [@student.42.fr](https://student.42.fr)

Résumé: *Ce projet est une première étape avant de créer votre propre Voxel World!*

Table des matières

I	Préambule	2
II	Introduction	4
III	Objectifs	5
IV	Consignes générales	6
V	Partie obligatoire	7
V.1	Le monde	7
V.2	Le rendu graphique	8
V.3	La caméra	8
V.4	Donc, en résumant :	8
VI	Partie bonus	9
VII	Rendu et peer-évaluation	10

Chapitre I

Préambule

By Mike Acton on March 14, 2008 9 :44 PM

One of the things we talked about this year at GDC was what we called the "Three Big Lies of Software Development." How much programmers buy into these "lies" has a pretty profound effect on the design (and performance!) of an engine, or any high-performance embedded system for that matter.

(LIE 1) SOFTWARE IS A PLATFORM

I blame the universities for this one. Academics like to remove as many variables from a problem as possible and try to solve things under "ideal" or completely general conditions. It's like old physicist jokes that go "We have made several simplifying assumptions... first, let each horse be a perfect rolling sphere..."

The reality is software is not a platform. You can't idealize the hardware. And the constants in the "Big-O notation" that are so often ignored, are often the parts that actually matter in reality (for example, memory performance.) You can't judge code in a vacuum. Hardware impacts data design. Data design impacts code choices. If you forget that, you have something that might work, but you aren't going to know if it's going to work well on the platform you're working with, with the data you actually have.

(LIE 2) CODE SHOULD BE DESIGNED AROUND A MODEL OF THE WORLD

There is no value in code being some kind of model or map of an imaginary world. I don't know why this one is so compelling for some programmers, but it is extremely popular. If there's a rocket in the game, rest assured that there is a "Rocket" class (Assuming the code is C++) which contains data for exactly one rocket and does rockety stuff. With no regard at all for what data transformation is really being done, or for the layout of the data. Or for that matter, without the basic understanding that where there's one thing, there's probably more than one.

Though there are a lot of performance penalties for this kind of design, the most significant one is that it doesn't scale. At all. One hundred rockets costs one hundred times as much as one rocket. And it's extremely likely it costs even more than that! Even to a non-programmer, that shouldn't make any sense. Economy of scale. If you have more

of something, it should get cheaper, not more expensive. And the way to do that is to design the data properly and group things by similar transformations.

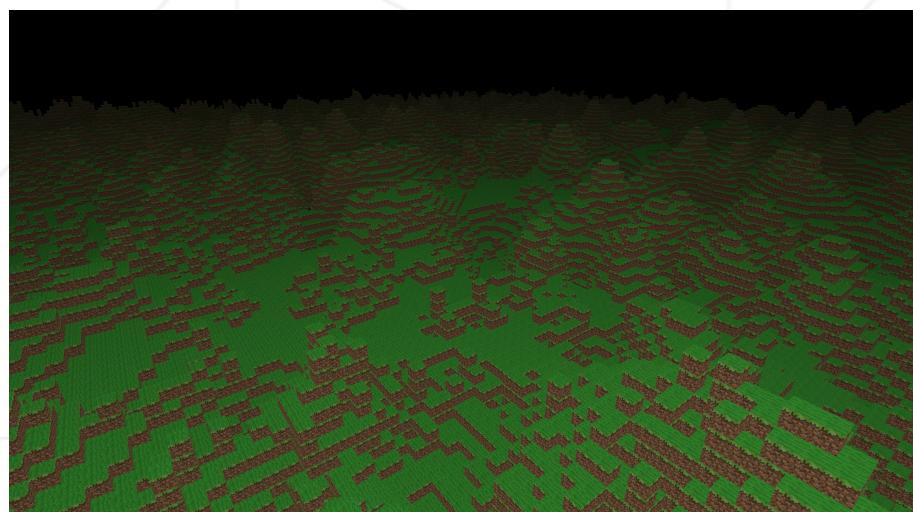
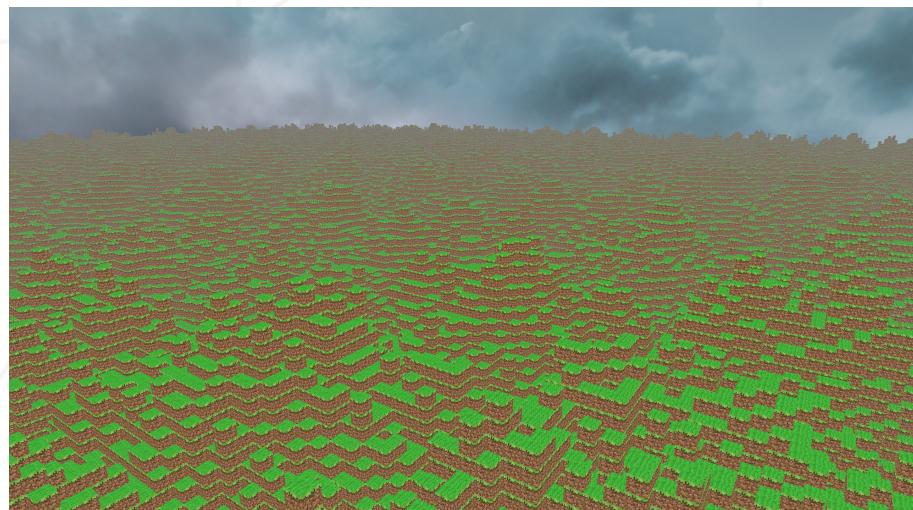
(LIE 3) CODE IS MORE IMPORTANT THAN DATA

This is the biggest lie of all. Programmers have spent untold billions of man-years writing about code, how to write it faster, better, prettier, etc. and at the end of the day, it's not that significant. Code is ephemeral and has no real intrinsic value. The algorithms certainly do, sure. But the code itself isn't worth all this time (and shelf space! - have you seen how many books there are on UML diagrams?). The code, the performance and the features hinge on one thing - the data. Bad data equals slow and crappy application. Writing a good engine means first and foremost, understanding the data.

Chapitre II

Introduction

Bienvenue dans le merveilleux monde des voxels, là où vous tirerez tous les bénéfices de l'abstraction "Le monde entier est une grille en 3 dimensions" dans l'objectif d'afficher un nombre titanique de choses à l'écran, voyager dans un univers procédural mastodontesque et un terrain de jeu profondément malléable. [C'est quoi des voxels ?](#)



Chapitre III

Objectifs

Ce projet a pour but de vous confronter à un projet graphique relativement exigeant en terme d'optimisation. Vous devrez donc étudier les caractéristiques des voxels-world, et les utiliser, avec vos connaissances en infographie, pour afficher énormément de choses à l'écran. Vous devrez donc étudier différents algo/opti pour avoir un rendu FLUIDE (et il y en a beaucoup). Vous devrez également gérer correctement votre mémoire et vos structures de données, afin de pouvoir voyager dans un univers très, très grand. Une fois cela accompli, vous serez prêt à passer au niveau supérieur avec le projet ft_minecraft, qui sera encore plus exigeant.

Chapitre IV

Consignes générales

- Vous êtes libre d'utiliser le langage que vous voudrez, mais faites tout de même attention aux performances de ce dernier (Si vous n'avez pas d'idée, le c/c++/rust sont recommandés).
- Vous devez utiliser openGL ou openCL pour les calculs GPU. Aucune librairie ne doit faire le travail à votre place.
- Vous avez le droit d'utiliser une librairie pour charger des objects en 3d et des images, une librairie de fenêtrage, et une librairie de mathématiques pour vos calculs de matrices/quaternions/vecteurs. Vous ne devez pas les push dans votre repo, mais écrire vos propres script de téléchargement/installation.
- Le rendu doit toujours être FLUIDE, et par FLUIDE, on entend que si le correcteur pense que jouer à votre jeu serait déplaisant, il peut vous mettre 0.
- Tous crash (Uncaught exception, segfault, abort ...) est éliminatoire.
- On doit pouvoir utiliser votre programme pendant des heures sans qu'il gobe toute la mémoire, et sans qu'il ralentisse. Attention à votre gestion de la mémoire, VRAM comprise.
- Votre programme doit tourner en full screen, interdiction de diminuer la taille du frame buffer par default.

Chapitre V

Partie obligatoire

V.1 Le monde

Vous devez être en mesure de créer un très grand monde de manières procédurales. Pour ce project-ci, on doit au moins pouvoir visiter $16384 \times 256 \times 16384$ cubes (256 c'est pour la hauteur).



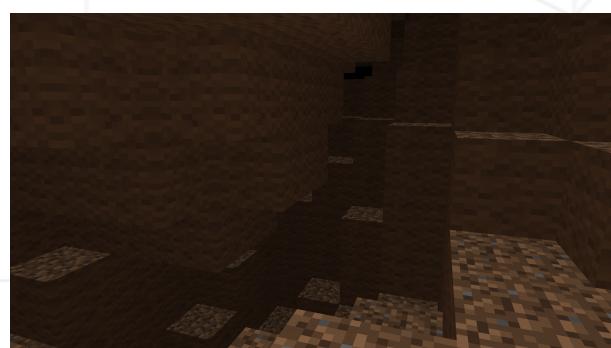
Pour le `ft_minecraft`, ce sera beaucoup, beaucoup plus gros.

Certains cubes peuvent être vides, et les autres peuvent être de type terre, herbe, sable etc ...

A part les cubes vides, ils seront tous opaques mais auront leurs textures propres. Vous devez implémenter une méthode pour générer un terrain comme des collines et des montagnes, et des caves lorsqu'on s'aventure en dessous du niveau du sol. Cette génération doit être déterministe, c'est à dire qu'avec un même seed la carte doit être exactement la même.

Le terrain doit avoir une topographie à peu près naturel, avec des collines et/ou des montagnes, des cavernes etc... un simple `rand()` n'est pas acceptable.

Chaque morceau de terrain visité doit être gardé en mémoire, jusqu'à une certaine limite que vous fixerez, ou vous pourrez commencer à supprimer des cubes déjà en mémoire.



V.2 Le rendu graphique

On doit également afficher les cubes à l'écran. La render distance minimal à ciel ouvert est de 160 cubes.

Pour vous donner une idée, elle est de 320 dans la première photo dans l'introduction. Bien sûr vous avez le droit de ne pas afficher des cubes s'ils sont cachés.

Chaque cube doit être texturé, et vous devez avoir au moins 2 textures et types de cubes différents. Débrouillez-vous pour que le FOV ait toujours l'air à peu près rempli, parcourir votre jeu (au sol) ne doit pas donner de résultats contre-intuitifs. Encore une fois, le rendu doit être fluide. Attention également aux freezes. Le FOV doit être de 80 degrés. Pour que ce soit un peu plus joli, vous devez aussi intégrer une skybox. Attention à ne pas laisser d'artifacts sur les jonctions de celle-ci.



Si vous voulez que votre rendu soit fluide, débrouillez vous pour que la charge de travail soit correctement partagé entre le CPU et le GPU.

V.3 La caméra

Si on peut voyager dans votre jeu, c'est bien aussi. Vous devez configurer une caméra sympathique. C'est-à-dire que la souris doit pouvoir la faire tourner sur au moins deux axes et que vous avez 4 touches qui permettront d'aller devant, derrière, à droite, à gauche, le tout étant relatif à la rotation de la caméra. Bien sûr, on doit pouvoir rester appuyé sur une touche pour avancer en continue. La vitesse de la camera doit être d'à peux près 1 cube par seconde, mais pour la correction, vous devez créer une touche spéciale qui permettra de multiplier la vitesse par 20.

V.4 Donc, en résumant :

- Un terrain gigantesque composés de cubes texturé de plusieurs types différents.
- Un FOV bien rempli.
- Une génération procédurale poussée qui donne au terrain un aspect naturel (collines, montagnes, cavernes ...).
- Une caméra intuitive.
- Une skybox.

Chapitre VI

Partie bonus

J'imagine que vous trépignez d'envie de rajouter de la physique, un joueur, des monstres verts qui réduisent en miettes des heures de travail, mais tout cela est réservé au projet ft_minecraft. Ici, on va se concentrer sur la partie technique.

- Avoir une render distance supérieure à 14 et être toujours fluide.
- Un compteur de fps, affiché a l'écran.
- Le rendu est toujours fluide et sans freezes, même en vitesse x20.
- Pouvoir supprimer des blocs avec la souris.
- Avoir de nombreux biômes différents.

Chapitre VII

Rendu et peer-évaluation

Rendez-votre travail sur votre dépôt **GiT** comme d'habitude. Seul le travail présent sur votre dépôt sera évalué en soutenance.