Guillaume ASSAILLY
Martin JOUEN
Romain KLODZINSKI
Nathan MEGUIN
Paul QUINZI

RAPPORT DE PROJET
MIICRAFT

Responsable Projet : Loic Yon

25/02/2024

Table des matières

Introduction....4
Abstract....4
Contraintes Techniques....5
Présentation du support....5
Wii....5
Environnement de programmation....6
DevKitPro....6
GX....6
Dolphin....6
Conception....7
Choix des Structures de données....7
Blocs....7
Chunks....7
Monde....7
Joueur....8
Caméra....9
Slot et Craft....9
Inventaire....9
Rendu....10
Présentation....10
Rendu 3D....10
Camera....10
Texture Environment Unit – TEV....11
Anomalie guirlande....13
Transparence....13
Lumières et Occlusion ambiante....14
Optimisation....18
Génération du monde....23
Génération des reliefs....23
Biomes....23
Structures....24
Strates et couches....25
Jouabilité....27
Physique....27
Interactions avec les blocs....27
Interfaces et inventaires....28

Organisation....29
Prévisionnelle....29
Réelle....31
Présentation du Résultat....33
Fonctionnalités....33
Génération infinie....33
Lumières....33
Physique....33
Commandes....33
Déplacements....33
Interactions avec l'environnement....34
Inventaires et Crafts....34
RSE....35
Conclusion....36
Tables des illustrations....37
Lexique de Minecraft....39
Bibliographie....40

Introduction

Pour notre projet de deuxième année, nous cherchions un sujet à l'aspect ludique qui permettrait de nous mettre dans des conditions de fortes contraintes pour la programmation.

C'est dans cet objectif que nous avons choisi de développer un jeu pour la Wii. Cette console de jeux sortie en 2006 par Nintendo a pour concept de proposer un catalogue de jeu familial et surtout adaptés à des commandes innovantes grâce à une nouvelle manette : la Wiimote, chargée d'accéléromètres et de gyroscope. La dernière mise à jour système de la console a eu lieu en 2012 et la documentation déjà rare sur son fonctionnement est de plus en plus enfouie dans les méandres d'Internet. La plupart des librairies propres au développement sur Wii sont aujourd'hui quasiment obsolètes, ou plutôt, leur développement est terminé depuis parfois plus de 10 ans. C'est pourquoi le choix d'un tel support nous permettrait de fournir un vrai travail d'ingénierie inversée et de recherches approfondies à la manière de celui que nous pourrions être amenés à faire dans le cadre d'un projet de migration de logiciel dans une entreprise utilisant encore des versions arriérées.

Viens ensuite le choix du jeu. Nous nous sommes là rapidement mis d'accord sur Minecraft. Un immense classique du monde du jeu vidéo sortie en 2011 par le studio suédois Mojang dans lequel le joueur évolue dans un monde ouvert et cubique. La renommé internationale de ce jeu l'a amené à être porté sur de très nombreux supports allant du PC à toutes les consoles de jeu moderne ainsi qu'à une version mobile. Seule une plateforme majeur contemporaine de Minecraft n'a pas eu le droit à sa propre version officielle : la Wii. De plus, les graphismes simplistes du jeu nous permettraient de développer un jeu en 3D en ajoutant à notre convenance des plus en plus de réalisme à nos graphismes et à nos interactions sans pour autant se lancer dans la conception d'un jeu aux graphismes déjà réaliste (en seulement trois semaines). Malgré le fait que le jeu soit originalement codé en Java, nous avons décidé d'appliquer nos connaissances de cette année en développant notre version en C/C++.

Vous trouverez dans ce rapport un bref aperçu des contraintes techniques liées au support et à l'environnement de programmation. Ensuite, nous aborderons la conception détaillée de chaque grand axe du projet. Nous discuterons par la suite de notre organisation pour parvenir à avancer dans un groupe de cinq personnes. Enfin, vous verrez un petit aperçu du résultat final !

Abstract

This report documents the development process of a Wii game as part of a second-year project, aimed at exploring challenging programming constraints in a playful manner. Utilizing the unique capabilities of the Wii console and its innovative controller, the Wiimote, our project involved reverse engineering and extensive research due to the lack of documentation and quasi obsolete development libraries. The chosen game, Minecraft, renowned for its open-world sandbox gameplay, presented an exciting challenge as it had not been officially ported to the Wii platform. Despite Minecraft's original Java codebase, we opted to implement our version in C/C++, utilizing our knowledge from this academic year. This report provides an overview of technical constraints, detailed project conception, group organization strategies, and a glimpse of the final outcome.

Contraintes Techniques

Présentation du support

Wii

Présentation

La Wii (prononcée « oui ») est une console de salon de 7ème génération conçu par Nintendo sortie en 2006, faisant suite à la Nintendo GameCube. Elle sera remplacée en 2012 par la Wii U.

La Wii a été une révolution dès sa sortie en proposant des façons de jouer uniques, grâce à sa nouvelle manette : la télécommande Wii ou Wiimote. En effet, cette manette a la particularité d'être dotée de technologies très avancées tels des capteurs de mouvements (gyroscope & accéléromètre) et caméra infrarouge, permettant ainsi de pointer l'écran et d'utiliser les mouvements du joueur en jeu, rendant l'expérience plus accessible et immersive. La simplicité de cette manette, de par son faible nombre de boutons, permet une prise en main aisée pour toutes les personnes, initiées ou non, et de tout âge, faisant de cette console familiale la 2ème console de salon la plus vendu au monde avec 102 millions d'exemplaires [1].

Figure 1 : Console Wii noire & Wiimote

Caractéristiques techniques

La Wii (*project Revolution*) est basée sur l'architecture de sa prédécesseure, la GameCube (*project Dolphin*) de 2001, avec un nombre limité d'améliorations, faisant de la Wii une console considérée comme en retard, sur le plan technique, d'une génération entière.

En effet, celle-ci possède un processeur monocœur vieillissant IBM PowerPC [2], ainsi que 88 Mo de mémoire vive GDDR3, répartit en 2 puces : MEM1 (24 Mo) et MEM2 (64 Mo). La partie graphique est quasi-identique à celle de la GameCube, avec un GPU limité en sortie à une résolution de 480 p maximum à 60 Hz.

Ces limitations sont dues à des choix marketing et philosophique durant la conception, privilégiant la façon de jouer (la Wiimote) à la puissance matérielle et au graphisme, permettant également de très fortement réduire le prix de vente.

Environnement de programmation

DevKitPro

Le développement logiciel sur Wii requiert l'utilisation d'un kit de développement logiciel (SDK) particulier propriétaire, fournis sur dossier, par Nintendo, aux partenaires : le Revolution-SDK ou RVL-SDK. Il n'est cependant plus possible d'en obtenir un exemplaire légalement aujourd'hui.

Afin de pouvoir développer sur Wii sans être affilié à Nintendo, nous utilisons un environnement de développement open-source, nommé DevKitPro. Il s'agit d'un SDK non-officiel utilisé par les communautés des « scènes homebrew » de chaque console Nintendo supportées (GameCube, Wii, Wii U, Switch, DS et 3DS). DevKitPro, qui n'est pas approuvé par Nintendo, permet ainsi une alternative légale au RVL-SDK officiel. Il est cependant à noter que l'exécution de code non-signé par Nintendo sur une vrai console requiert son piratage, que celui-ci soit développé avec DevKitPro ou le RVL-SDK. Le piratage, dans le cas présent, n'enfreint pas les lois en vigueur en France, concernant la propriété intellectuelle, et est donc considéré comme légal.

GX

Le développement de ce projet sera effectué en C++ 20, avec l'utilisation de l'IDE CLion, cmake, git (GitHub) et de DevKitPro à travers Windows Subsystem Linux (WSL).

Hormis le SDK, le développement et la programmation d'un jeu Wii sont identiques à tout autres types de logiciel en C++, à l'exception du rendu graphique. En effet, sur ordinateur ou consoles de jeu plus récentes, le développement graphique, notamment de graphisme 3D est géré par des bibliothèques logicielles standards et bas-niveau comme DirectX, Vulkan ou OpenGL, ou encore des moteurs graphiques plus haut-niveau tels qu'Unreal Engine ou encore Unity, facilitant grandement le développement. Cependant, ces technologies ne sont pas disponibles sur Wii, qui possède sa propre bibliothèque graphique propriétaire : GX (Revolution Graphics Library).

GX est l'interface de contrôle du processeur graphique de la Wii (nommé GP), permettant de lui envoyer des commandes de rendu graphique et autres données relativement simplement. Si GX ressemble à OpenGL, celui-ci reste cependant à un niveau d'abstraction bien plus bas-niveau et proche du matériel.

Il est également à noter que l'implémentation de GX fournis par DevKitPro n'est pas rigoureusement identique à celle de RVL-SDK, souffrant du manque de certaines fonctionnalités ainsi que d'une documentation, nous forçant à procéder à de l'ingénierie inverse afin d'en déterminer le fonctionnement et les bonnes pratiques et optimisations. Cette ingénierie inverse n'ayant, par moment, plus suffit, nous avons dû utiliser les documentations officielles confidentielles de GX [3] et du RVL-SDK [4], trouvable sur Internet, pour nous aider à comprendre le fonctionnement de l'architecture interne de la console.

Dolphin

Afin de faciliter le développement, nous avons utilisé Dolphin. Il s'agit d'un émulateur à destination de la GameCube (d'où son nom) et de sa petite sœur, la Wii, nous permettant de tester facilement et rapidement pendant le développement, plutôt que d'utiliser une vraie Wii pour chaque test. La Wii a cependant été utilisée pour des tests importants de performances, de validation fonctionnelles et d'utilisation de la Wiimote.

Conception

Choix des Structures de données

Blocs

Dans la version actuelle du projet, un bloc contient uniquement un *BlockType,* soit une énumération contenant l'intégralité des blocs disponibles dans notre jeu. L'ordre des blocs dans cette énumération permet grâce à des balises de créer des macros grâce auxquelles nous pourrons connaître des propriétés du bloc voulu notamment si celui-ci est transparent ou semi-transparent, s'il est d'une forme particulière et beaucoup d'autres possibilités. Notre structure nous permet déjà de stocker des blocs non posables comme des pioches ou des composants de fabrication qui n'existes pas encore dans notre version. La classe bloc n'existe pas seulement pour être une surcouche de *BlockType,* mais aussi pour pouvoir y ajouter des informations comme la résistance du bloc au minage sans alourdir le poids d'un chunk (en utilisant *BlockType* comme identifiant).

Chunks

Un chunk est le nom donné aux subdivisions du monde dans Minecraft. Son nom dans l'arborescence : *verticalChunk* est un vestige de notre première structure de donnée. En effet, nous avions dans un premier temps récupéré la notion des chunks horizontaux existante dans Minecraft. Cependant, nous avons rapidement fait le choix de nous séparer de cette surcouche, car elle n'intervenait ni dans les optimisations de rendu que nous pensions mettre en place, ni dans les fonctions du back-end ou en tout cas sans amélioration.

Ces chunks, ou chunks verticaux sont donc des ensembles de 16 blocs par 16 sur 128 blocs de haut que nous stockerons dans des tableaux de *BlockType* en trois dimensions. Ils comportent un identifiant unique. Nous avons également choisi de donner la conscience à chaque chunk de ses voisins notamment pour faciliter la génération de structures, la gestion des lumières ainsi que certaines optimisations de rendu sans passer par la map du monde dont nous parlerons plus tard. Chaque chunk connaît également ses coordonnées dans cette map. Enfin, pour rester sur les lumières, les chunks embarquent une file de lumières permettant de tenir compte des sources de lumière artificielle comme des lampes.

Monde

Comme nous l'avons déjà abordée précédemment. Notre monde est avant tout constitué d'une map de chunks. Au-delà du jeu de mots, cette map monde contient l'ensemble des chunks chargés avec pour clé leurs coordonnées dans le monde. Cela permet d'avoir une correspondance facile entre les identifiants des chunks et leurs coordonnées et d'avoir à la manière d'une mappemonde, une vision vue du terrain vu du haut. Voici par exemple une carte du relief de 100 chunks générés. L'altitude n'est pas directement stockée dans world, mais cette figure permet de visualiser le système de coordonnées de chunks.

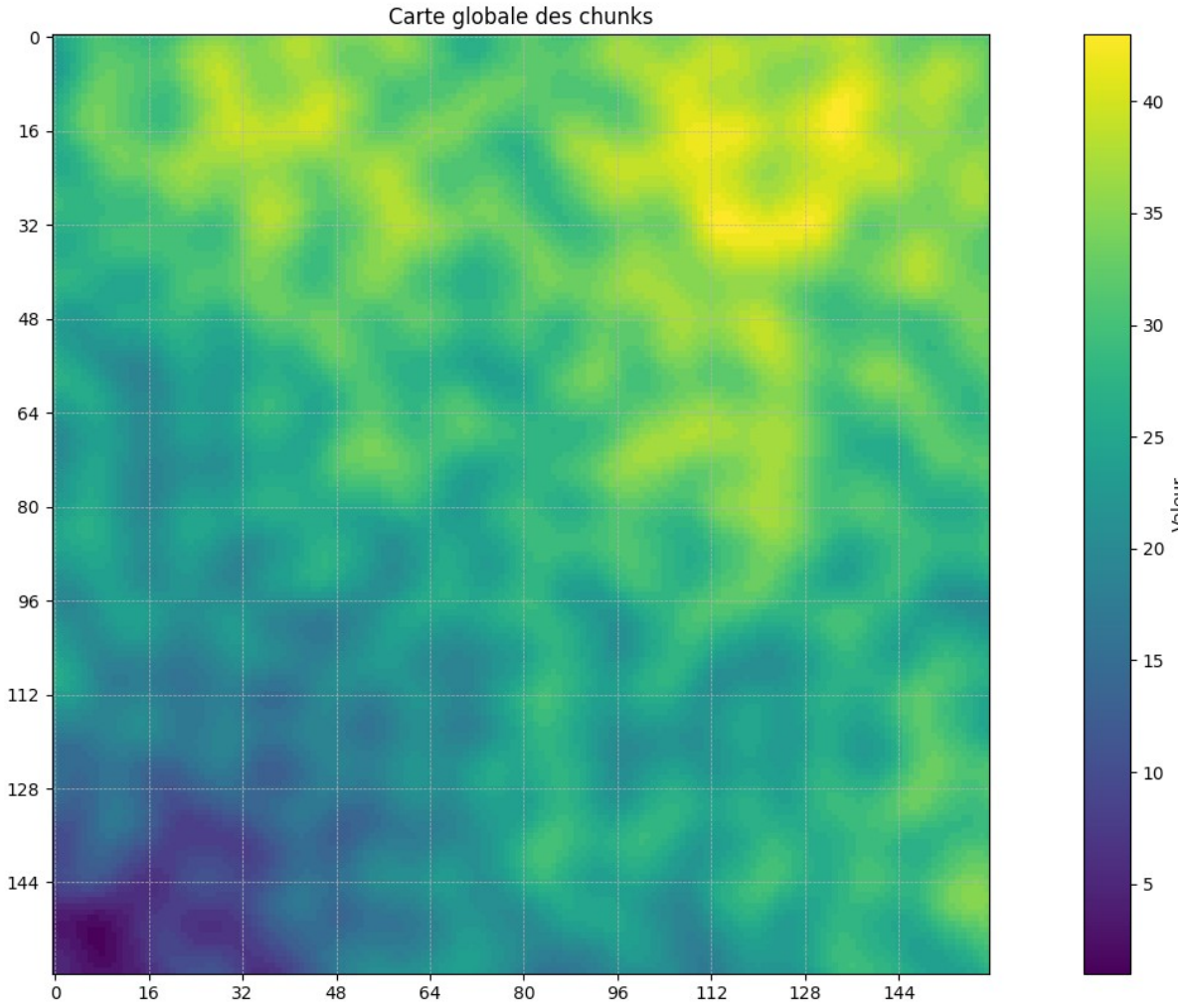


Figure 2 : Carte des chunks

Nous stockons également un set de chunks contenant uniquement les coordonnées référant aux chunks générés mais pas encore chargé. Cela permettra d'avoir connaissance des chunks stockées dans la mémoire (carte SD) mais pas encore chargés dans la RAM.

Joueur

La classe Joueur est une classe majeure de notre projet puisqu'elle embarque un grand nombre de fonctionnalités. On y retrouve pour cela un certain nombre de booléens permettant d'obtenir un certain nombre d'informations sur le joueur telles que le fait qu'il soit sensible à la gravité, le fait qu'il sprint ou encore le fait que ce joueur soit accroupi. Certaines de ces valeurs peuvent être changées en jeu tandis que les autres sont réglables dans le code à des fins de débogage ou dans un futur menu. Un joueur a également connaissance de son inventaire (que nous détaillerons plus tard). Le joueur est responsable de son mouvement et de sa propre gravité avec des valeurs d'accélération et de vitesse verticales qui lui sont propre. Cette classe contient aussi un certain nombre d'informations sur les blocs qu'il cible avec sa caméra. Justement, le joueur est indissociable de la caméra avec laquelle sa position est extrêmement couplée.

Caméra

La caméra d'un jeu 3D sur Wii avec GX possède une forme assez standard contenant le champ de vision vertical est horizontal et surtout trois vecteurs up, pos et look représentant respectivement un vecteur pointant vers le haut depuis la caméra, la position de la caméra est celle du point que l'on regarde. Nous ajouterons à cela les paramètres de la distance de rendu. Les fonctions permettant l'affichage de la 2D sont aussi dans la caméra, car elles correspondent à une transformation orthogonale des matrices pour que les vertex semblent s'afficher sur l'écran.

Slot et Craft

Le *Slot* est la classe qui représente les groupes d'objets de même type utiles pour la gestion de l'inventaire et de la fabrication. Un slot est composé d'un type d'objet ainsi que de sa quantité dans ce slot. Cette quantité sera limitée par les fonctions à un stack, soit 64 unités comme dans le jeu original.

Craft regroupe quant à lui des listes de neufs slots associés à des résultats à la manière d'une recette de cuisine. C'est par cette structure que passe l'intégralité de la fabrication.

Inventaire

La classe *Inventory* contient toute la gestion de l'inventaire du joueur ainsi que celle des interfaces de craft (ou fabrication). L'inventaire en lui-même est représenté par un tableau de slots avec un nombre de ligne de 4 en mode survie et potentiellement infini en mode créatif, et 9 colonnes. La première ligne représente systématiquement la « hotbar » ou barre d'accès rapide. C'est cette barre qui sera en permanence affichée en bas de l'écran et depuis laquelle nous pouvons poser des blocs. Cela permet de développer une sorte de pagination de l'affichage de l'inventaire sans altérer cette barre d'accès rapide. Un slot supplémentaire correspondra au slot sélectionné par le curseur et affiché à côté de celui-ci.

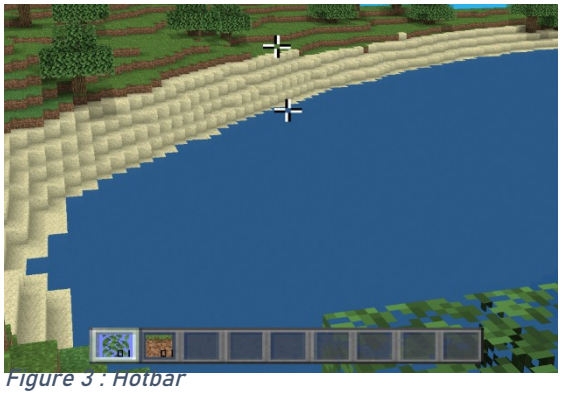


Figure 3 : Hotbar

Figure 4 : Inventaire

Rendu

Présentation

Le moteur de rendu du jeu est basé sur GX, il est au cœur du programme et requiert le plus de puissance possible. En effet, c'est lui qui rend/génère les graphismes et les frames (images rendues) en collaboration avec le processeur graphique GP. Il nécessite une attention particulière tout au long du développement afin de garantir des performances maximales et le frame rate (taux d'images rendues par seconde : 60 fps max = 60 images par seconde) le plus haut et stable possible.

Rendu 3D

La première étape dans le développement du moteur de rendu est de pouvoir rendre de la 3D simple, ici, de simples cubes. Un GPU en général ne connaît que des primitives simples tels que des lignes, des triangles, qui sont l'association ordonnée de 3 points (vertex) dans l'espace, ou des quadrilatères (quads), assemblage de 2 triangles. Afin de rendre les 6 faces d'un cube, nous avons donc besoin de 6 quads de 4 vertex chacun, soit 24 vertex pour un simple cube.

Cependant, il est déjà nécessaire de prendre en compte les limites du GPU. En effet, ce dernier ne peut rendre que 40.5 millions de vertex par secondes [3], soit 675 000 vertex par frame à 60 fps, soit seulement 28 125 cubes, c'est-à-dire moins qu'un seul chunk rempli.

Camera

La caméra est essentielle. C'est elle qui permet de savoir ce qui est dans le champ de vision et sous quels angles sont vu les objets. La caméra est modélisée par 3 vecteurs : sa position dans l'espace, un vecteur indiquant le haut (up) et la direction de la caméra (look). Ces vecteurs sont ensuite assemblés sous une forme matricielle, utilisable par le GPU.

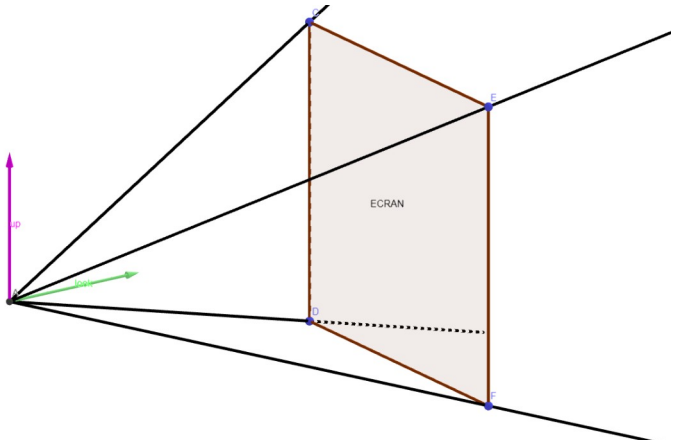


Figure 5 : Caméra 3D (point A) et cône de projection

Une matrice de projection est également utilisée, c'est ce qui nous permet entre autres de définir l'angle de vue (field of view – FOV) de la caméra et de projeter les vertex 3D dans l'espace de l'écran 2D. Il en existe 2 types prédéfinis en GX :

- Matrice de projection perspective : principalement utilisée pour du rendu 3D, elle permet un effet de perspective en déformant les vertex éloignés du centre de l'écran.

- Matrice de projection orthogonale : principalement utilisée pour du rendu 2D, elle ne prend pas en compte la distance des objets et les présente de face, à plat, sans perspective, nous l'utilisons pour le rendu de l'interface utilisateur et des menus.

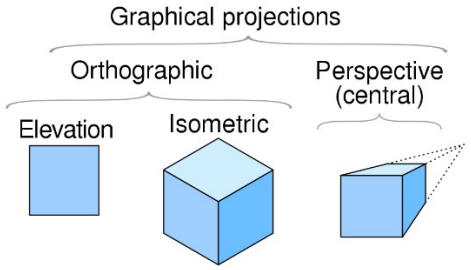


Figure 6 : Types de projection spatiale [5]

Texture Environment Unit – TEV

GX a été conçu initialement pour la GameCube, fin des années 1990. Or, à cette époque, la technologie des shaders programmables n'existait pas encore. Un shader est un programme exécuté par le GPU afin de faire, relativement facilement, des traitements sur l'image générée, tels que des calculs de lumière, d'ombre, de mixage (*blending*) et de texture. On retrouve principalement 2 types de shader :

- Les vertex shader (traitant les vertex : lumières / couleurs),
- Les fragment shader (traitant les pixels : mixage et texturage).

Bien que cette technologie ait commencé à se populariser dès le début des années 2000 avec l'apparition de la référence dans le domaine, GLSL (OpenGL Shader Language), elle n'est pas présente sur Wii non plus, afin de garder une architecture simple.

En lieu et place des shaders programmables, se trouve donc le Texture Environment Unit (TEV), une technologie propriétaire beaucoup plus primitive que les shaders, mais redoutablement efficace pour l'époque. Le TEV correspond au fragment shader, l'équivalent du vertex shader étant déjà effectué par le hardware du GPU (rastérisation et calcul de lumière). Le TEV consiste en l'application d'une simple formule mathématique figée, pour chaque pixel de l'image, prenant en paramètre 4 nombres A, B, C, D. La programmation du TEV consiste à spécifier, à travers GX, quelles valeurs doivent être transmises à ces paramètres. Il est possible d'exécuter plusieurs fois le TEV avec des paramètres différents, dans la limite de 16 fois (*16 stages*). Le résultat de chaque itération étant réutilisable par l'étage suivant.

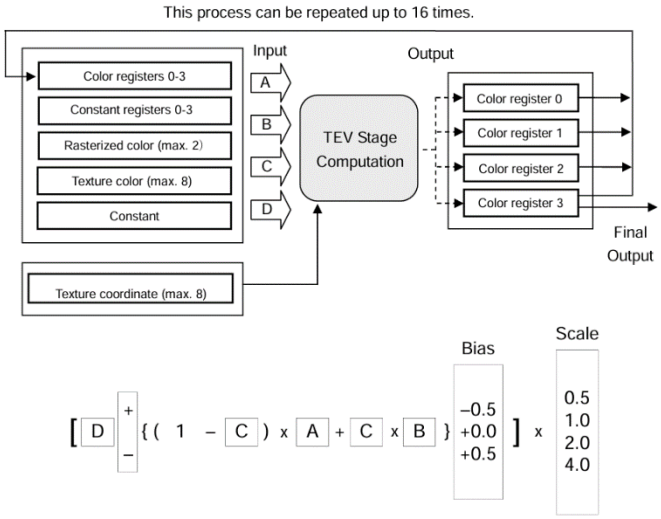


Figure 7 : Schéma du TEV & formule de calcul [6]

Dans notre cas, nous avons simplement besoin d'utiliser un seul stage, pour mixer la couleur rastérisée des objets avec leur texture. La couleur rastérisée est calculée à partir de la couleur intrinsèque des vertex (que nous utilisons pour notre propre système d'ombre et lumière), et des sources de lumière (que nous n'utilisons pas car trop limitées pour nos besoins). Ainsi, notre configuration du TEV prendra cette forme :

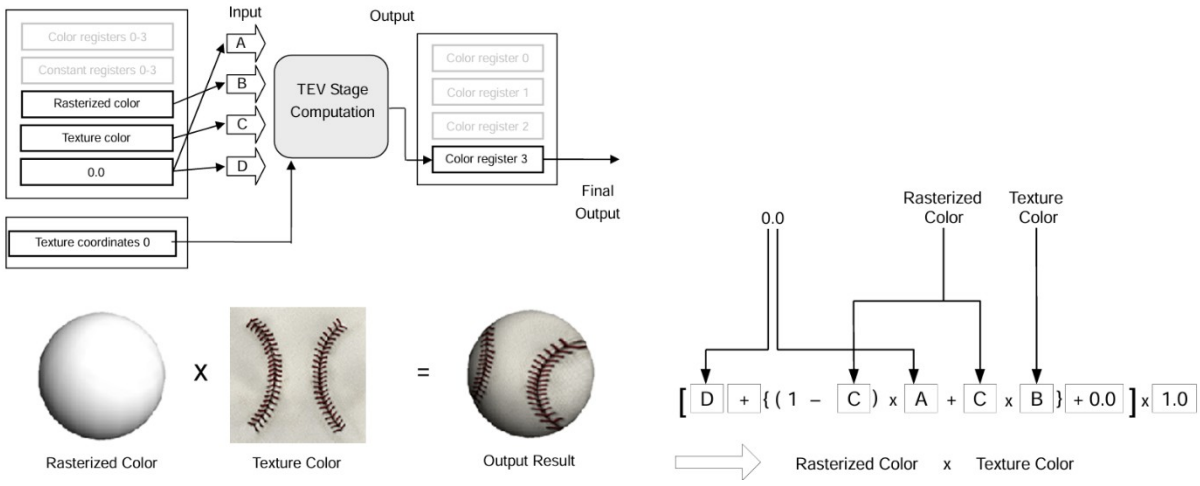


Figure 8 : Configuration du TEV en texturing modulated [6]

Il est donc possible de jouer avec la teinte, la saturation et l'opacité d'une texture, simplement en modulant la couleur intrinsèque (et donc la couleur rastérisée) des vertex. Ci-dessous, nous testons à gauche, des variations autres que le blanc (couleur rastérisée neutre) sur la texture de base à droite (que nous détaillerons plus tard).

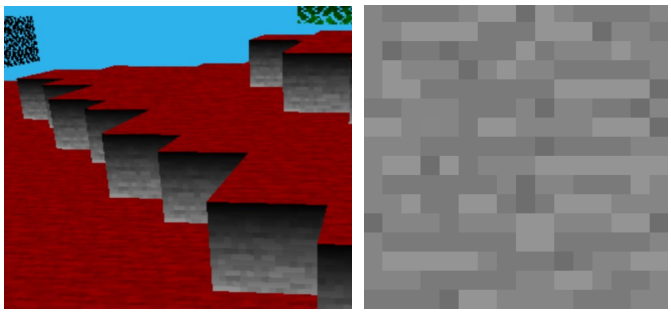


Figure 9 : Tests de modulation et PoC sur la texture de pierre.

Anomalie guirlande

L'anomalie guirlande est le nom que nous avons donné à une erreur de corruption de la texture survenu de façon prévisiblement aléatoire, lors des débuts du moteur de rendu. La corruption prend toujours la forme de guirlande verte horizontale, bien que d'autres espèces de guirlandes, plus rares, sous forme de grilles et matrices, aient existé.

L'anomalie a pu être esquivée en mettant en place des conventions de codage visant à réduire son apparition à néant, avant d'être définitivement résolue, suite à l'utilisation des documentations officielles du RVL-SDK. L'anomalie provenant d'un défaut d'alignement mémoire, pouvant survenir par simples modifications de code indépendant dans le projet.

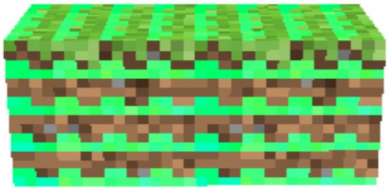


Figure 10 : Espèce de guirlande la plus commune (~98% des rencontres)

Transparence

Maintenant que nous sommes capables de rendre des textures sur des objets, une attention particulière est requise pour les textures présentant une composante transparente. En effet, lors du rendu, le GPU utilise un Z-buffer, il s'agit d'un système permettant, lors du rendu d'un pixel, de savoir sa profondeur sur l'image. Ainsi, cela permet de ne pas écraser le rendu d'un objet situé devant un autre, si ce dernier est rendu avant. Cependant, ce système ne fonctionne plus si la couleur rendue est transparente. Si nous rendons un fond marin, puis la mer (transparente), alors rendre quelque chose entre le fond marin et la mer devient impossible, car l'eau est considérée comme devant. Cela implique que l'ordre de rendu des objets doit être pris en considération afin de réaliser un rendu correct, les objets transparents devant donc être rendu en dernier.

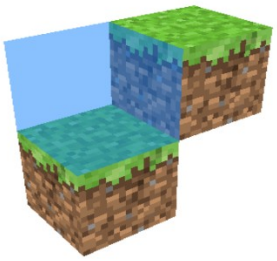


Figure 11 : R&D sur la transparence

Lumières et Occlusion ambiante

Un aspect fondamental du rendu, insoupçonné au départ et qui semblait également assez anodin est celui des lumières. Effectivement, passé un certain stade de développement, lorsque le rendu semblait de plus en plus mature, le problème de l'éclairage nous est peu à peu apparu.

Les lumières et les ombres (cela va de pair) offrent plusieurs avantages :

- Une meilleure compréhension de la topologie et la géométrie du terrain : en effet, dans un jeu à paysage cubique, si des faces avec la même texture ont des valeurs de lumière différentes, l'œil humain est beaucoup plus enclin à se représenter correctement l'environnement, évitant un bon nombre de problèmes de perspectives.
- Obscurcir les zones non touchées par la lumière naturelle : soyons réaliste, dans un jeu où l'objectif principal est de miner dans des grottes, on peut au moins s'attendre à ce que la lumière s'adapte à un environnement obscur. La lumière, en plus d'embellir le rendu, est un élément de *gameplay* à part entière, des sources de lumière naturelles comme le soleil aux sources plus artificielles comme les torches ou les lampes.

Il semblait donc évident d'accorder une grande partie de notre développement à ces questions.

Essais sur le matériel

La première idée envisagée pour les lumières, était d'utiliser les objets lumineux directement intégrés à la librairie GX. Ces sources lumineuses, après quelques réglages comme leur position, couleur, type (ambiante, spotlight, etc.), permettent au GPU de calculer automatiquement des ombres sur les objets opaques environnants. Cette solution, qui semblait à première vue la plus pratique et la plus simple, s'est avérée être la moins concluante :

- D'abord par limitation technique : une scène rendue par GX ne peut pas prendre en compte plus de 8 lumières à la fois. Même si la raison matérielle derrière nous semble assez « obscure », il n'empêche que c'est assez limitant, voire rédhibitoire si on doit implémenter un système de lumières artificielles.
- Nous avons également rencontré plusieurs problèmes techniques : impossibilité d'implémenter automatiquement des *shadowmaps* (systèmes d'ombres projetées) ou encore l'utilisation de *shaders* récents.
- Enfin, le rendu final de ces premiers tests n'étaient tout simplement pas représentatifs d'un semblant de Direction Artistique propre à Minecraft.

Figure 12 : Test des ombres dynamiques GX

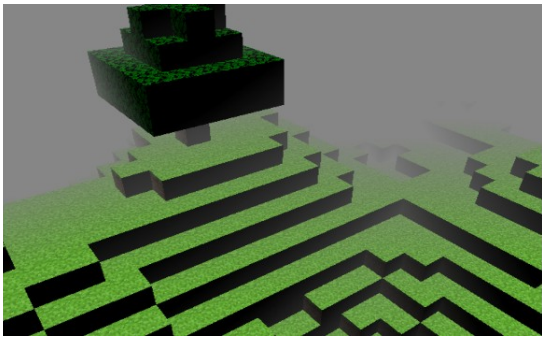


Figure 13 : Test de lumière de type spotlight

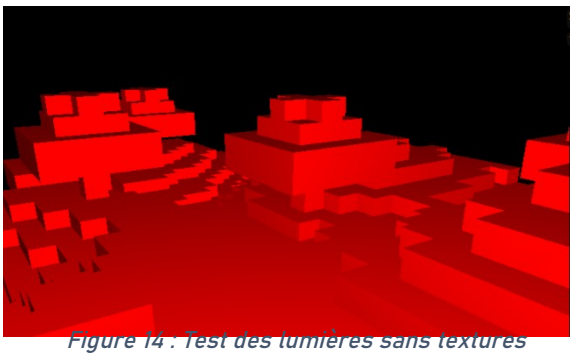


Figure 14 : Test des lumières sans textures

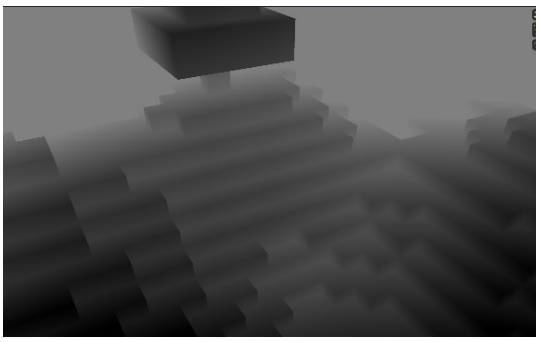


Figure 15 : Le brouillard comme outil d'occlusion

C'est à ce moment-là que nous est apparu l'étendu de notre solitude dans un océan de techniques inexplorées. Rien n'existait à priori pour calculer des ombres automatiquement, il nous faudrait donc implémenter ces algorithmes nous-mêmes.

Zone d'ombres

Minecraft fonctionne avec un système un peu barbare mais efficace : le jeu dispose de 16 niveaux de lumières, de 0 à 15, 0 étant la valeur la plus sombre, et 15 la lumière du soleil. En termes de gameplay, et indépendamment du rendu, ces valeurs permettent de contrôler l'apparition de certains monstres (à certains seuils de luminosité) ou encore à opérer un cycle jour/nuit. Dans les premières versions de développement du jeu, celles qui nous ont du moins servies de modèle, ces valeurs impactent le rendu de manière brutale : chaque face de bloc possède une valeur de luminosité, et elle est assombrie en conséquence, 0 étant l'assombrissement maximal, 15, la texture normale du bloc. Pour ce cas d'école, il existe un algorithme récursif connu de propagation de lumière.

C'est pourquoi nous avons décidé de partir sur cette mécanique : en plus d'être plus facile à implémenter (principalement algorithmique), nous nous assurions un rendu proche d'un Minecraft *vanilla*.

Première problématique : comment stocker les valeurs lumineuses de chaque face dans nos chunks, sans perdre trop d'espace mémoire (une denrée rare) ? La plupart des projets de clones utilisaient des cartes de lumières en plus du stockage des blocs, mais cela était bien trop volumineux.

La solution était plus simple :

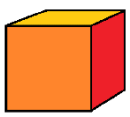


Figure 16 : Un bloc

Considérons le cube ci-dessus : supposons qu'il s'agisse d'un cube solide à la surface du terrain disposant de plusieurs valeurs de lumières : respectivement 15 sur la face du dessus, 14 sur la face avant, 13 sur la face droite. Cette situation est équivalente au schéma suivant :

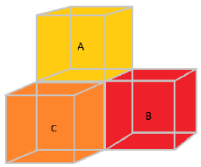


Figure 17 : Le même bloc

Où A, B, et C sont des blocs d'airs contenant une seule valeur de luminosité. Cette solution permet de calculer pour chaque face d'un bloc solide son niveau lumineux en fonction du bloc d'air voisin, sans avoir à stocker une seule autre structure de donnée, on ne conserve d'ailleurs pas de valeurs de luminosité redondantes pour des blocs internes (ici, cela ne s'applique logiquement qu'aux blocs en surface, donc en contact avec des blocs d'airs). Le seul inconvénient est de gérer maintenant non pas 1, mais 16 types de blocs d'airs différents. Les algorithmes de propagation lumineuse du jeu original pourront être réutilisés et adaptés à ce mécanisme de bloc d'air. En voici une version très simplifiée :

```
Soit alpha la valeur de luminosité d'un bloc.
Procédure InitLight() :
Pour chaque bloc y de 127 à 0 :
    Pour chaque bloc x de 0 à 15 :
        Pour chaque bloc z de 0 à 15 :
            Coordonnées p = {x, y, z}

            Si le bloc à la position p dans le chunk est un bloc d'air :
                Si y est égal à 127 :
                    Mettre le bloc à la position p à une luminosité maximale alpha(p) = 15
                    Ajouter p à la file d'attente de lumière du chunk
                Sinon :
                    Mettre le bloc à la position p à une luminosité minimale alpha(p) = 0


Procédure PropagateLight() :
Tant que la file d'attente de lumière du chunk n'est pas vide :
    Prendre le premier élément p de la file d'attente

    Pour chaque voisin v de p :
        Si alpha(v) < alpha(p) - 1
                alpha(p) =  alpha(p) - 1
                alpha(v) = alpha(p) - 1 // Sauf si il s'agit du voisin du dessous (la lumière du soleil est
infinie verticalement)
                Ajouter v à la file d'attente.
```

Il reste à prendre en compte quelques éléments comme les sources de lumière artificielle, la propagation de la lumière actualisée entre les chunks, et à chaque modification du terrain par le joueur, etc.

Se pose ensuite la question de la coloration des textures : il est évidemment impossible et non négociable de stocker 16 versions différentes de chaque texture de bloc correspondant à un degré de luminosité. Il faut que cette coloration n'ait lieu qu'au moment du rendu, dynamiquement en fonction du bloc d'air le plus proche.

C'est pourquoi nous avons réutilisé le principe de l'environnement de Texture de GX pour actualiser les couleurs de chaque face. En fonction du bloc d'air environnant, on applique l'une des 16 couleurs de modulation.

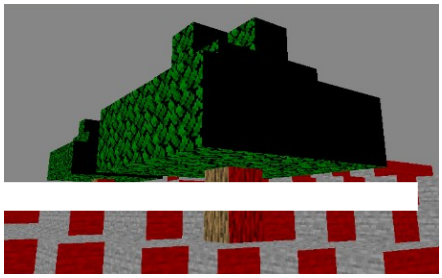


Figure 18 : Preuve de concept du TexEnv

Sur l'exemple ci-contre, nous hard-codons chaque vertex de chaque face nord à une couleur rouge : on ne garde donc que la composante rouge de la texture, ce qui explique que les feuilles qui en sont dépourvues apparaissent noires.

Fort de cette preuve de concept, on peut maintenant appliquer le nuancier suivant, en parallèle de notre algorithme de propagation, et obtenir des résultats très satisfaisant.

Figure 19 : Nuancier des 16 niveaux d'ombre

Occlusion ambiante

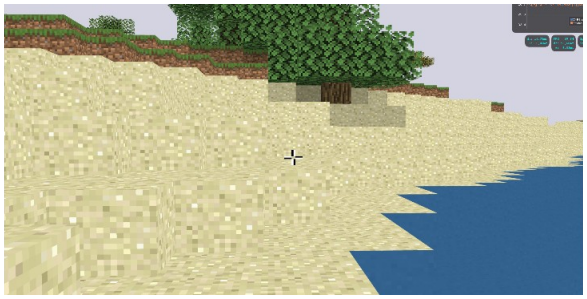


Figure 20 : Rendu sans occlusion ambiante

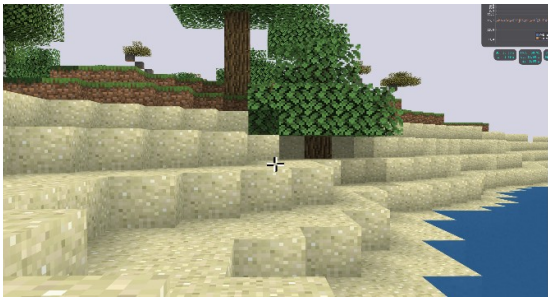


Figure 21 : Rendu avec occlusion ambiante

L'occlusion ambiante est une technique d'ombrage qui simule l'obscurité résultant de l'occlusion des surfaces environnantes, c'est-à-dire une zone moins susceptible de recevoir le rebondissement de la lumière. Cet effet à première vue anodin, est central en imagerie 3D et un des aspects les plus importants de traitement de *shading*. C'est également un élément fondamental pour faire comprendre à l'œil quels volumes sont visibles.

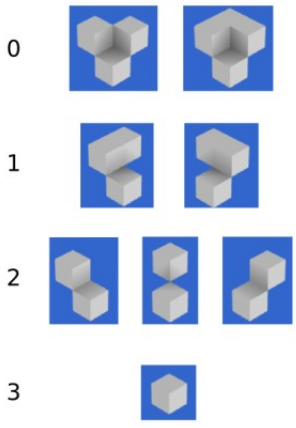


Figure 22 : Patrons d'occlusion ambiante

Là encore, pas de solutions magiques, tout doit être implémenté à la main. L'avantage d'un jeu cubique, c'est que le nombre de patterns d'occlusion est assez limité, et résulte d'un principe assez simple.

Chaque face d'un cube comporte quatre vertex. Il suffit de fixer la couleur de chaque vertex parmi l'une des trois suivantes : blanche, noire ou grise. Cette attribution de couleur dépend du contexte (le voisinage) de la face (voir la liste des cas ci-contre).

Figure 23 : Attribution des couleurs aux vertex d'une face

Voici un exemple d'attribution de couleur par vertex. On utilise la même technique d'environnement de texture que pour les ombres présentées plus haut. Pour cette affectation, au rendu, l'anti-aliasing du GPU transforme cette texture en un dégradé beaucoup plus détaillé :

Figure 24 : Face d'occlusion après lissage

Calquée sur la texture du bloc, on obtient ainsi, avec un peu d'algorithmie, des effets d'occlusion ambiante. Plus encore que les ombres, cette technique a été la plus payante pour nous approcher du rendu du jeu original.

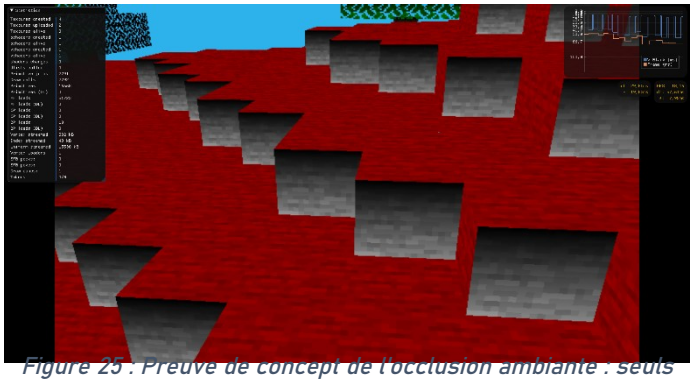


Figure 25 : Preuve de concept de l'occlusion ambiante : seuls les deux vertex supérieurs ont été noircis

Note : pour rendre compatible cet effet avec celui des ombres, nous ne pouvons plus utiliser le nuancier précédent, il faut donc 3 niveaux de couleurs par ombre, les ombres blanches, grises et noires, soit 3 x 16 = 48 couleurs. Cette palette reste encore acceptable.

Optimisation

Comme présentées rapidement en introduction, les performances sont un enjeu majeur du développement du moteur de rendu, qui a connu de nombreuses transformations et améliorations durant sa conception. En effet, le GPU est faible, même pour son époque, ce qui requiert des choix importants d'optimisation, énoncés ci-après dans l'ordre chronologique d'implémentation.

Back-face culling

Le back-face culling est la première optimisation à avoir été mise en place, étant directement gérée et prise en charge par le GPU (sur activation). Cela consiste à donner les vertex des polygones (triangles ou quads) dans un ordre précis. En effet, de cette façon, si le GPU lit les vertex d'un polygone dans un sens horaire, alors il considère que celui-ci est vu de face et le rend. Sinon, il considère qu'il s'agit de la face arrière du polygone et annule son rendu. Cette optimisation primordiale permet de ne pas rendre l'intérieur des objets, que nous ne sommes pas censés vouloir voir. Cela nous permet dans notre cas de diviser directement le nombre de

polygones rendu par 2, ce qui n'est pas négligeable, étant donné que les performances sont ainsi considérées comme doublées.

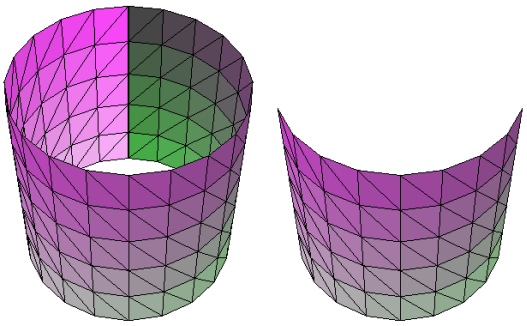


Figure 26 : cylindre sans (à gauche) et avec (à droite) culling [7]

Adjacence culling

Ne plus rendre l'intérieur des cubes est essentiel, mais cela ne fait pas de miracle, nous rendons toujours trop de cubes. Pour cela, nous avons mis au point un algorithme simple mais redoutablement efficace. L'idée est de ne prendre que « la surface », le contour d'un ensemble de cubes et non plus chacun d'eux. Cela peut en réalité se faire très simplement en vérifiant la présence de cubes voisins à chaque face d'un cube donné en cours de rendu.

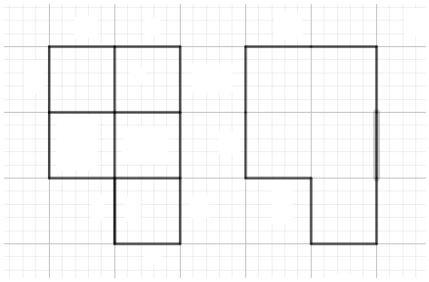


Figure 27 : exemple sans filtrage (à gauche) et avec (à droite) en 2D

Cette optimisation immensément efficace a permis un gain de performance estimable à environ 6 000 %, passant d'environ 1 fps pour 1 chunk à 60 fps pour 3 x 3 chunks.

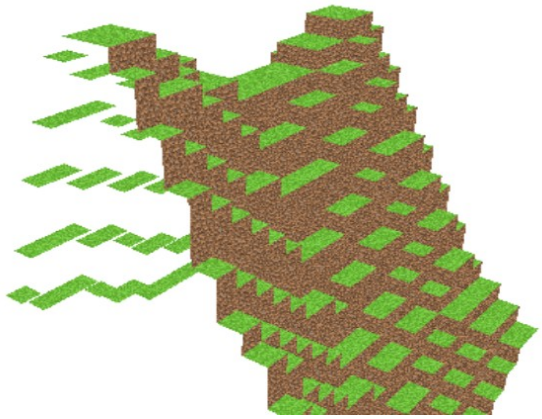


Figure 28 : surface d'un chunk rendu avec l'adjacence culling

Accès indirect & optimisation du pipeline

Cette optimisation est plus pratique qu'algorithmique. Elle consiste à utiliser GX au mieux. En effet, l'envoi de commande au GPU se fait à travers un canal (ou pipeline FIFO), dans lequel nous envoyons entre autres des matrices de transformation de coordonnées de vertex, et des vertex.

Les matrices de transformation (*modelview matrix*) permettent d'avoir un objet à plusieurs endroits possible sans changer les coordonnées dans les vertex, qui sont donc des relatives. Elles sont essentielles dans un rendu normal, mais inutiles et trop lourdes dans notre cas. En effet, dans un rendu standard, le terrain est statique et les objets dynamiques sont peu nombreux. Or, dans notre cas, l'entièreté du terrain est dynamique, ce qui implique la présente de matrice pour chaque cube rendu. Nous pouvons supprimer aisément l'intégralité des matrices en passant les coordonnées des vertex en coordonnées absolues. L'impact a été une amélioration globale des performances de 300 %.

Une autre optimisation en lien avec le pipeline est de réduire la quantité de données envoyée. En effet, par défaut, les données sont envoyées en mode directe, c'est-à-dire directement dans le pipeline. Ce qui pour notre format de vertex nous donne :

- Coordonnées 3D x, y, z (en flottant) : 12 octets
- Coordonnées 3D des normales : 12 octets
- 1 couleur RGBA8 : 4 octets
- Coordonnées 2D de texture (u, v) : 8 octets

Soit 36 octets par vertex. GX permet l'envoi de données en mode indirecte indexé, ce mode permet d'envoyer seulement un identifiant/indice de correspondance, de 1 ou 2 octets, sur une table déclarée au préalable. Par exemple, comme nous ne rendons que des cubes, nous n'avons que 6 valeurs de normales différentes, ce qui peut aisément être indexé, tout comme la couleur dans notre cas. Par ailleurs, suite au développement de notre propre système de lumière utilisant la couleur, nous n'avons plus besoin des normales. Nos vertex sont donc de forme :

- Coordonnées 3D x, y, z (en flottant) : 12 octets
- 1 couleur RGBA8 indexée : 1 octet
- Coordonnées 2D de texture (u, v) : 8 octets

Soit 21 octets par vertex. L'impact, bien que positif, n'as pas pu être déterminé. En effet, les performances sur console réelle ne pouvant être mesurées, nous utilisons le framerate indiqué par l'émulateur Dolphin, qui est peu sensible à ce type d'optimisation dans les programmes émulés. Il reste cependant à noter que ce gain d'espace sera déterminant plus tard lors de la mise en cache.

Frustum culling

Le frustum culling est une technique consistant à ne pas rendre les objets en dehors du champ de vision, nous l'avons implémentée à l'échelle des chunks, en lien avec la dernière optimisation de mise en cache. Ainsi, nous ne rendons que les chunks au moins partiellement visible (en bleu ci-dessous). Cela permet une optimisation théorique estimable à environ 300 % (dépend du champ de vision, l'angle et la position de la caméra, et la distance de rendu).

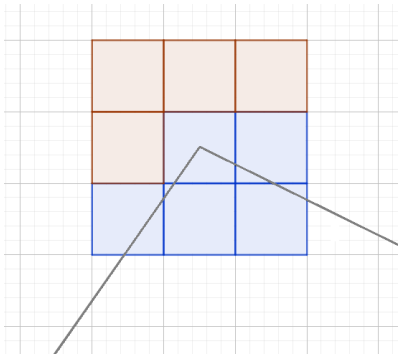


Figure 29 : chunks visible en bleu dans le cône de vision

Cache de pré-rendu

Il s'agit certainement de l'optimisation la plus puissance mise en place jusqu'ici. L'idée est de ne plus re-rendre la même chose à chaque frame. Plus précisément, nous utilisons une mécanique de GX, qui consiste à rendre 1 fois un objet, non pas dans le pipeline principal, mais dans des pipelines secondaires appelés des display lists. L'idée est donc de calculer une seule fois les données des vertex et de les mettre de côté, prête à être envoyé au GPU. Nous effectuons cette optimisation à l'échelle des chunks, afin d'avoir le moins de display lists active (et les plus remplis) possible, tout en étant capable de les régénérer rapidement.

Si cette optimisation peut paraître plutôt simple à mettre en place, il est nécessaire de mettre en lumière les principales contraintes :

- Le rafraîchissement du cache : il y a besoin de savoir, à tout moment, si un chunk a été modifié depuis son dernier pré-rendu, de façon à le pré-rendre à nouveau, pour éviter la désynchronisation de l'état interne et de ce qui est rendu à l'écran.
- La taille du cache : rappelons que nous pouvons théoriquement espérer rendre 675 000 vertex par frame, à 21 octets par vertex, cela nous donne une taille de cache de 13.5 Mo, ce qui est beaucoup par rapport à nos 88 Mo de RAM (cette taille peut varier si nous considérons utile de pré-rendre des zones finalement non rendues).
- La gestion des display lists : il s'agit du problème principal que nous allons détailler par la suite.

Tout d'abord, d'un point de vue technique, l'allocation mémoire du cache ne peut se faire que de manière statique, 1 seule fois, pour des raisons évidentes de temps d'allocation. Il vient ensuite le problème de la gestion des display lists. En effet, ces dernières sont contiguës et leurs tailles ne sont pas connues à l'avance. Il est donc impossible de gérer efficacement le cache après un certain nombre de réactualisation de pré-rendu.

Pour régler ce problème, nous segmentons l'espace de cache contiguë en un nombre arbitraire de slots, de display lists de tailles forcées fixes. Ainsi, le pré-rendu d'un chunk peut couvrir un certain nombre de ces petites display list, et ce, non nécessairement dans un ordre séquentiel. Par ailleurs, nous associons à chaque liste un type, indiquant s'il s'agit du rendu de cube transparent ou non, et ainsi envoyer au GPU ces listes en deux passages : opaque et transparent.

L'impact de cette optimisation n'est pas quantifiable, tant il y a de paramètres à prendre en compte. Cependant, nous sommes aisément parvenus à élever la distance de rendu à 10 chunks de distance (160 m) au lieu de 3 chunks (48 m).

Figure 30 : exemple minimal de ce que permet le cache

En outre, cette méthode de rendu ouvre la voie à l'implémentation d'un mode multijoueur en écran partagé, sans surcoût sur les performances. En voici ci-dessous la preuve de concept :

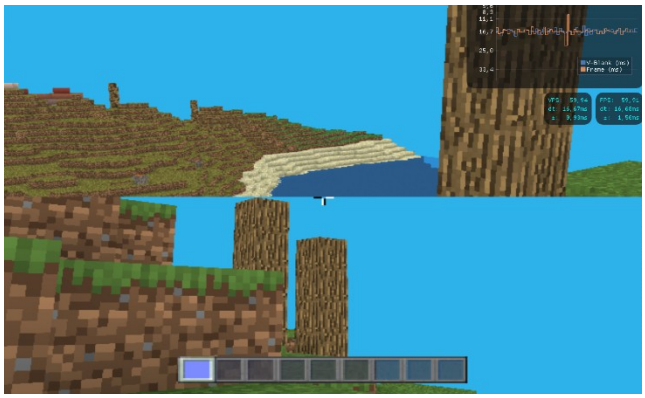


Figure 31 : PoC du mode 2-joueur en splitscreen

Génération du monde

L'objectif de cette partie est de former un monde pour le joueur qui ne soit pas prévisible lors de son exploration, et qui reste cohérent et aussi réaliste que possible.

Des bruits de gradients (Perlin, OpenSimplex, Value) seront utilisés pour satisfaire ces critères. Ils ont pour avantages de mixer aléatoire et cohérence, contrairement à l'aléatoire pur qui est bien trop chaotique pour cette utilisation. Chaque algorithme crée une forme de bruit différente, qui sera choisie et ajustée selon le résultat voulu [8]. Les méthodes de génération du monde sont grandement inspirées du jeu réel, bien que souvent simplifié car notre projet est limité par les performances de la console, ainsi qu'en temps (et donc en fonctionnalités) [9].

Génération des reliefs

La première étape consiste à générer le relief du monde sans les structures (arbres et autres), ni le climat ou le type de blocs présent. Cela se traduit par trouver la hauteur de chaque colonne de blocs.

Pour cela, on utilise trois bruits de paramètres différents qui seront échantillonnés au point précis de la colonne de blocs à générer. Les trois bruits ont notablement une échelle différente, et auront un impact plus ou moins fort sur la hauteur des blocs. En détails, on peut noter :

- Le bruit continental, qui permet de former des océans et des plages. Il a une forte influence sur le niveau du fond marin, et beaucoup moins dès que l'on génère un continent. Il permet également une cohérence du niveau de la mer (uniforme sur le tout le monde), en évitant des cas où une plage passerai sous le niveau de la mer.
- Le bruit d'érosion indiquera si le terrain est plat (donc une plaine, un désert) ou montagneux. Il va agir comme un coefficient sur le bruit suivant et détermine la forme du terrain à grande échelle.
- Enfin le bruit altitude, qui viendra former les collines, pics montagneux et petits reliefs. C'est le bruit avec les détails les plus petits des trois.

Chaque bruit va passer dans un filtre afin de changer la réparation des valeurs extrêmes. Cela permet de mettre en valeur les différents reliefs et de les contraster. Cela est nécessaire pour s'adapter à la distance de vue du joueur qui ne permet pas de distinguer des reliefs plus réalistes de faible variation.

Cette méthode de génération est très personnalisable et peut être améliorée en rajoutant d'autre bruits, mais ne permet pas de surplomb rocheux par exemple. Il aurait fallu utiliser un bruit 3D pour cela.

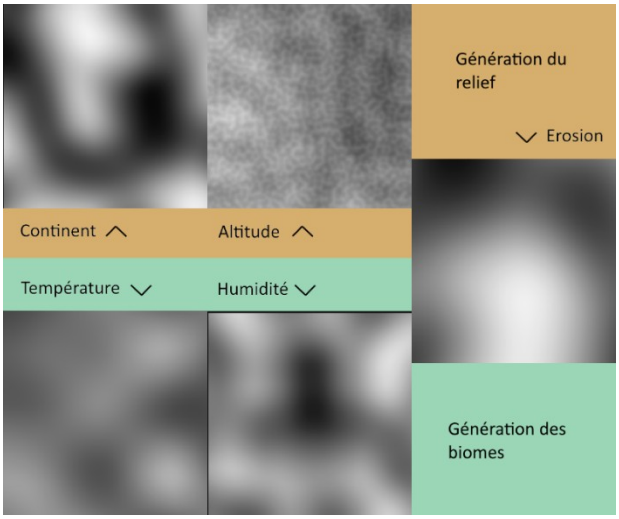


Figure 32 : Différentes couches de bruits

Biomes

Il faut maintenant peindre le terrain avec différents types de paysages (appelés « biomes »). Nous allons donc utiliser la même méthode que pour le relief.

- Le bruit de température, viendra décider de générer une toundra à la place d'une plaine, ou un désert si sa valeur est haute. Ce bruit permet d'avoir une cohérence du climat, et évitant d'avoir un désert à côté d'une toundra.
- Le bruit d'humidité indique la quantité de végétation de la zone. Il rentrera aussi en compte dans la génération des arbres.
- Ainsi que le bruit d'érosion généré précédemment pour le relief.

Selon la combinaison de ces trois paramètres, le biome d'une colonne de blocs est choisi et on place les blocs correspondants au biome en conséquence. Parmi les biomes disponibles, il est possible de croiser des déserts, des savanes, des jungles, des pics glacés, des toundras, des taïgas, des plaines, des mesas, des montagnes rocheuses, ainsi que des variations plus ou moins boisées.

Pour la génération de la mer, dès qu'une hauteur de colonne est inférieure au niveau de la mer qui est fixé (environ 30 blocs de haut), de la mer est générée. De la même manière, un seuil de hauteur détermine les plages, en prenant quand même en compte les autres bruits pour un meilleur rendu. Cette méthode est la plus simple pour avoir des côtes cohérentes.

L'inconvénient est de ne pas pouvoir faire de lacs à plus haute altitude, mais cela n'est pas gênant de par l'échelle assez petite des montagnes et continents.

Structures

On place maintenant les différentes structures sur le monde, s'agissant surtout d'arbres. Placer des structures de plusieurs blocs de surface n'est pas un problème facile à résoudre, couplé à la génération chunk par chunk. En effet, les structures à cheval sur plusieurs chunk sont problématiques, car l'un des chunks n'as pas encore été généré. Il faut aussi prendre en compte que le terrain est incertain dans ce chunk non généré, car un mur ou une falaise pourrait se trouver dans le chemin de la structure. Cependant, nous ne générons que de petites structures (moins d'un chunk de surface), et donc des simplifications sont possibles. La méthode la plus simple pour les arbres est de ne les générer qu'au centre des chunks. Comme l'on connaît à l'avance le rayon de l'arbre, il est possible de prévoir une zone en bordure de chunk, où aucun arbre ne sera généré.

Une amélioration de cette méthode vise à modifier cette zone interdite selon que le chunk voisin existe déjà, car on peut construire sur celui-ci. Cette amélioration permet d'éviter de créer des lignes sans arbres entre les chunks, qui est plutôt visible dans les forêts par exemple. Cette amélioration n'a pas été finalisée cependant, par manque de temps.

Le nombre d'arbre par chunk dépends de la moyenne d'humidité sur tout le chunk, ainsi que de la proportion de chaque biome. Donc un chunk avec 50 % de forêt classique, et 50 % de taïga, aura cette même proportion de chaque type d'arbres réparti sur le chunk sans tenir compte de la position des biomes. De cette manière, les arbres seront mélangés et la bordure des biomes sera plus fluide.

Voici des images montrant différents paysages et biomes du jeu.

Figure 33 : Biome Savane

Figure 34 : Biomes Taïga et Toundra

Figure 35 : Montagnes et Conifères

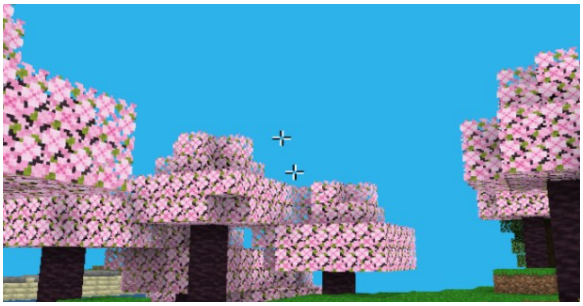


Figure 36 : Forêt de cerisiers

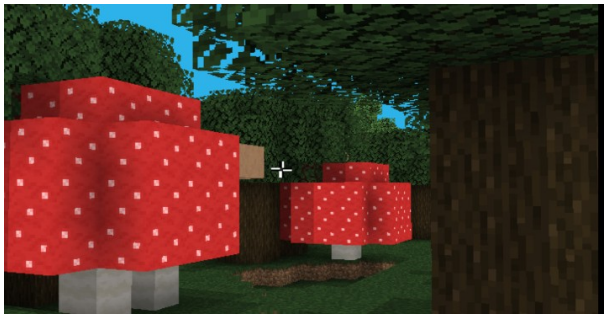


Figure 37 : Forêt sombre

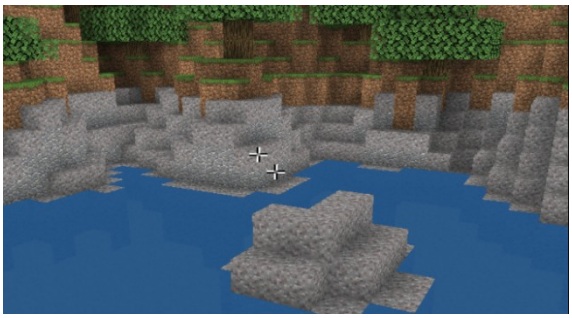


Figure 38 : Carrière de gravier

Figure 39 : Une mesa boisée

Strates et couches

Au-delà de la génération de surface, une recherche a également été faite sur les différentes couches de profondeurs. Notre version du jeu implémente les couches de bases du monde allant de la Bedrock avec une altitude aléatoire à la surface (souvent d'herbe) en passant par une grande couche de pierre et une plus fine de terre ou de sable. Il sera également possible à l'avenir d'ajouter des grottes voire des filons de minerais, mais il nous a paru plus intéressant de se concentrer sur la génération de surface pendant les trois semaines qui nous étés impartis.

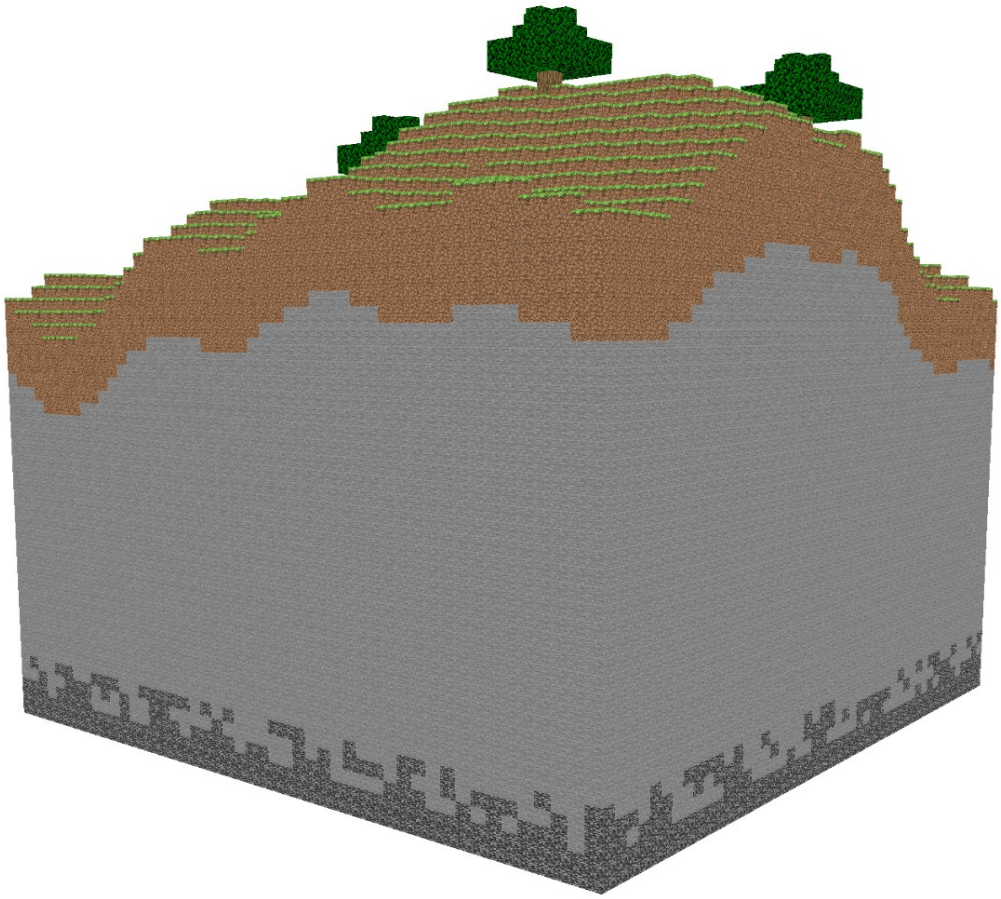


Figure 40 : Couches de générations

Jouabilité

Physique

Dans Miicraft, le joueur incarne un personnage de 1,80 mètre évoluant dans un environnement où chaque bloc mesure 1 mètre par 1 mètre. Le personnage lui-même occupe un espace de 30 centimètres en largeur et en longueur. Ces dimensions sont essentielles pour définir les interactions physiques, notamment les collisions avec les blocs. Le monde du jeu se veut réaliste, avec la présence de la gravité qui affecte le déplacement du personnage, mais pas les blocs. Ce dernier peut effectuer des sauts, représentant une simple impulsion verticale. Toutefois, l'eau se comporte de manière distincte puisqu'elle n'est pas solide : bien que soumise à la gravité, le personnage peut y pénétrer et nager.

Afin de faciliter les déplacements dans les endroits restreints, nous avons dû séparer le vecteur de mouvements en deux composantes x et z. Cela permet d'éviter de passer à travers deux blocs placés en diagonales et de bloquer le déplacement seulement sur l'axe où celui-ci est entravé.

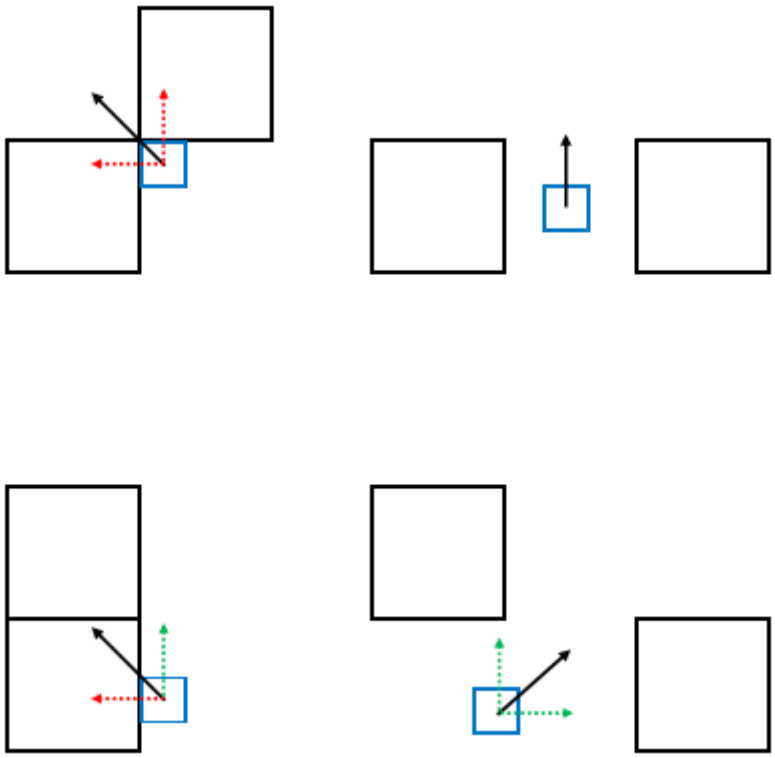


Figure 41 : Exemple collisions

Sur cet exemple, on peut voir le joueur en bleu, le vecteur de mouvement en noir, et les composantes en pointillé sont vertes si le joueur se déplace sur cette composante et rouge dans le cas contraire.

Interactions avec les blocs

L'essence même du jeu repose sur les interactions avec les blocs. Le monde offert au joueur est entièrement manipulable : chaque bloc peut être détruit, collecté et replacé à volonté. Ces possibilités permettent de créer des structures complexes. Les joueurs peuvent construire des maisons, des châteaux, des villes entières, et même des reproductions de monuments réels.

Le joueur dispose d'une portée de 5 mètres pour cibler un bloc spécifique. En visant un bloc, le joueur a la possibilité de le détruire, ce qui le récupérera dans son inventaire, ou de poser un bloc sur la face ciblée, consommant ainsi le bloc qu'il tient en main. Pour faciliter ces actions, le joueur peut également se concentrer sur un bloc, ce qui ajuste automatiquement la caméra pour que le joueur regarde ce bloc, indépendamment de ses propres mouvements.

Interfaces et inventaires

Figure 42 : Exemple de bloc visé

Le jeu offre une vaste gamme de blocs ainsi que la possibilité de fabriquer des objets et des blocs, ce qui requiert une interface ergonomique pour utiliser ces fonctionnalités de manière fluide. La "hotbar" permet de sélectionner rapidement le bloc à utiliser. Dans l'inventaire, les joueurs peuvent prendre un bloc ou un objet et le déplacer ailleurs grâce à une fonction de "drag and drop". Il est également possible de fabriquer des blocs dans l'inventaire, mais uniquement s'ils peuvent être créés dans une zone de fabrication de 2 par 2. Des raccourcis pratiques sont disponibles pour prendre la moitié des blocs, en poser un seul, ou déplacer rapidement un ensemble de blocs. Cette fonction de déplacement rapide permet de passer aisément entre la hotbar et l'inventaire, facilitant également la fabrication simultanée de plusieurs objets et leur transfert dans l'inventaire.

Une autre interface, la table de fabrication, offre une zone de fabrication plus grande de 3 par 3. Pour y accéder, les joueurs doivent interagir avec un bloc de table de fabrication, qu'ils peuvent eux-mêmes fabriquer dans la zone de l'inventaire.

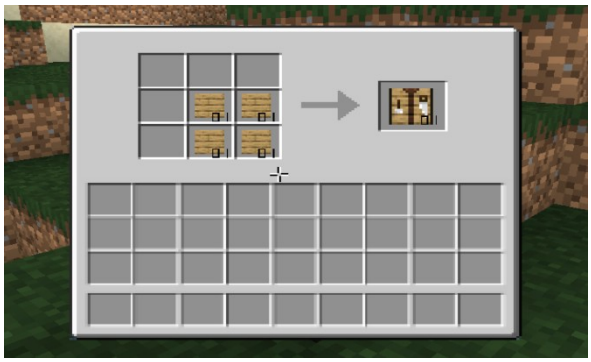


Figure 43 : Table de fabrication

Organisation

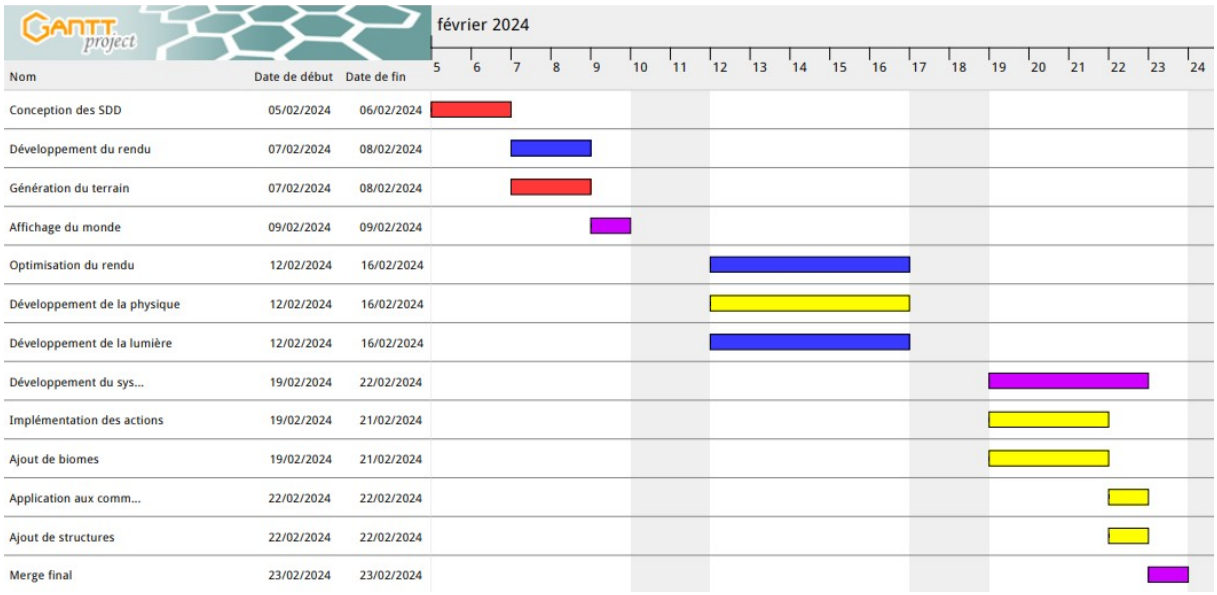
Ce projet représente une quantité de travail conséquente puisqu'il ajoute à la programmation énormément de recherche de documentation et de rétro-ingénierie sur des systèmes quasiment obsolètes de nos jours. Nous n'étions pas trop de cinq pour relever ce défi. Toutefois, cela demande une organisation particulière à laquelle peu d'entre nous avaient déjà été confrontés. Voici donc un résumé de notre organisation au cours de ces trois semaines intenses.

Prévisionnelle

Il est important de dire que l'objectif initial était de pouvoir rendre un monde de taille limité avec une génération procédurale du terrain et de pouvoir interagir avec cet environnement. Pour cela, nous avons commencé par mettre au point ce diagramme de Gantt prévisionnel ainsi qu'une répartition des charges primitive pour se donner un axe de départ.

Nom	Date de début	Date de fin
Conception des SDD	05/02/2024	06/02/2024
Développement du rendu	07/02/2024	08/02/2024
Génération du terrain	07/02/2024	08/02/2024
Affichage du monde	09/02/2024	09/02/2024
Optimisation du rendu	12/02/2024	16/02/2024
Développement de la physique	12/02/2024	16/02/2024
Développement de la lumière	12/02/2024	16/02/2024
Développement du système de sauvegarde et cache	19/02/2024	22/02/2024
Implémentation des actions	19/02/2024	21/02/2024
Ajout de biomes	19/02/2024	21/02/2024
Application aux commandes de la Wiimote	22/02/2024	22/02/2024
Ajout de structures	22/02/2024	22/02/2024
Merge final	23/02/2024	23/02/2024

Figure 44 : Liste des tâches prévisionnelles



Nom	Date de début	Date de fin
Conception des SDD	05/02/2024	06/02/2024
Développement du rendu	07/02/2024	08/02/2024
Génération du terrain	07/02/2024	08/02/2024
Affichage du monde	09/02/2024	09/02/2024
Optimisation du rendu	12/02/2024	16/02/2024
Développement de la physique	12/02/2024	16/02/2024
Développement de la lumière	12/02/2024	16/02/2024
Développement du sys...	19/02/2024	22/02/2024
Implémentation des actions	19/02/2024	21/02/2024
Ajout de biomes	19/02/2024	21/02/2024
Application aux comm...	22/02/2024	22/02/2024
Ajout de structures	22/02/2024	22/02/2024
Merge final	23/02/2024	23/02/2024

Figure 45 : Diagramme de Gantt prévisionnel

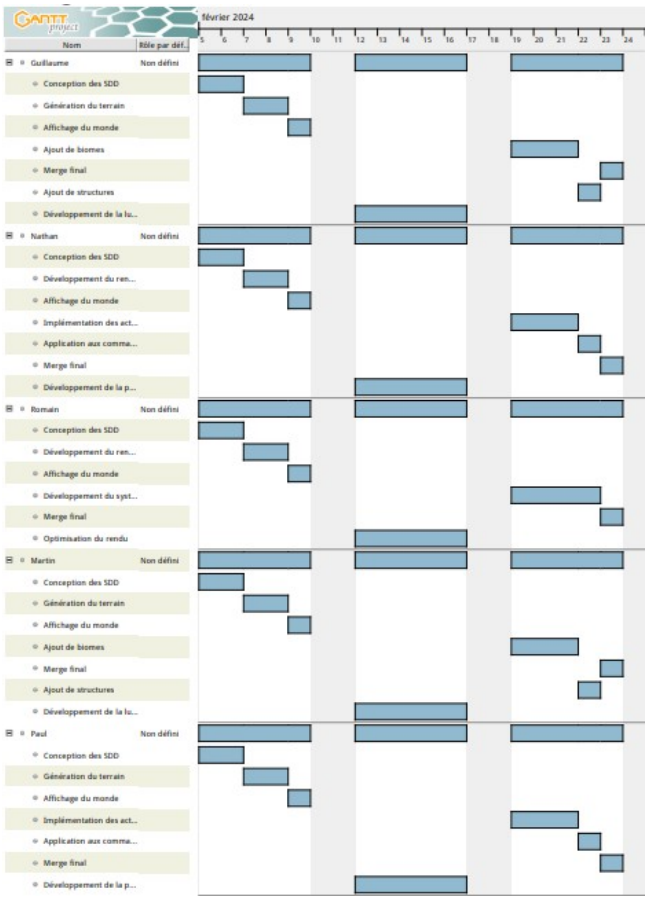
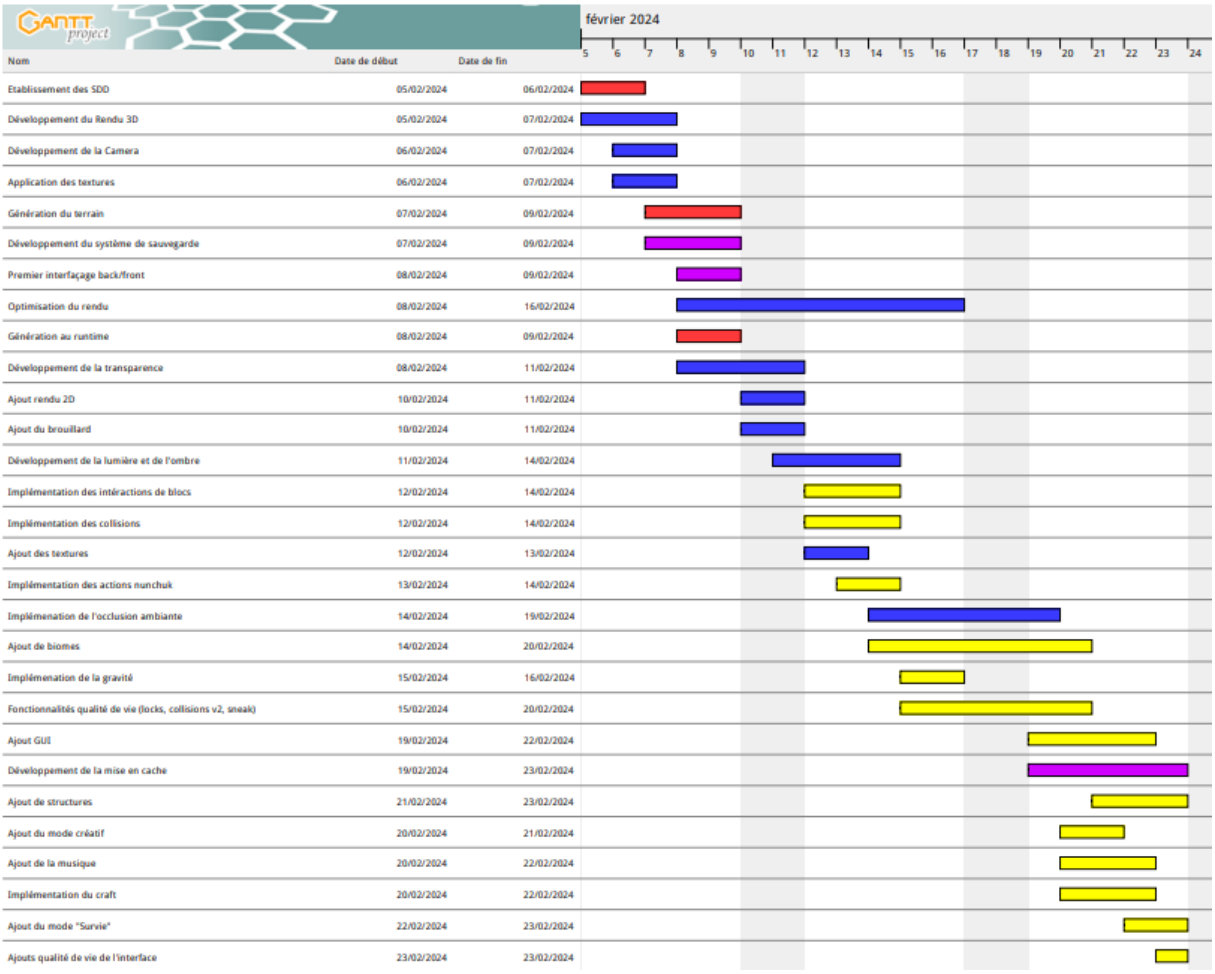


Figure 46 : Attribution prévisionnelle des tâches

En ce qui concerne la répartition, nous pouvons voir que la première semaine est constituée de tâches importantes et modulaires pour lancer le projet. C'est pourquoi nous avions choisi de nous mettre sur ces tâches en parallèle. Ensuite, la plupart des travaux sont assignés à des petites équipes de deux même si certaines tâches très spécifiques comme l'optimisation du rendu sont gérées par une seule personne afin que celle-ci puisse assimiler plus rapidement les spécificités des algorithmes. Enfin, la dernière semaine se concentrera surtout sur les interactions avec le monde ainsi que les ajouts à la génération du monde, et même à un système de sauvegarde sur lequel une seule personne sera affectée pour la même raison que précédemment. Il est évidemment plus simple de travailler à plusieurs sur le développement de plusieurs structures en parallèle que sur un système de sauvegarde monolithique.

Réelle

Premièrement, nous nous sommes fixé un rythme de travail en présentiel à 5 entre 8h30 et 18h30 quotidiennement, ainsi que quelques recherches durant les week-ends. C'est sans doute cette rigueur qui nous a permis d'avancer efficacement ensemble et d'éviter les conflits de versions. Pour cela, nous avons évidemment utilisé Github pour simplifier le travail sur les features sans détruire l'existant. Grâce à cela, nous pouvons voir que notre avancée a rapidement dépassé nos attentes malgré les difficultés rencontrées en raison du support. Cela nous a conduits à créer de nouvelles missions.



Nom	Date de début	Date de fin
Etablissement des SDD	05/02/2024	06/02/2024
Développement du Rendu 3D	05/02/2024	07/02/2024
Développement de la Camera	06/02/2024	07/02/2024
Application des textures	06/02/2024	07/02/2024
Génération du terrain	07/02/2024	09/02/2024
Développement du système de sauvegarde	07/02/2024	09/02/2024
Premier interfaçage back/front	08/02/2024	09/02/2024
Optimisation du rendu	08/02/2024	16/02/2024
Génération au runtime	08/02/2024	09/02/2024
Développement de la transparence	08/02/2024	11/02/2024
Ajout rendu 2D	10/02/2024	11/02/2024
Ajout du brouillard	10/02/2024	11/02/2024
Développement de la lumière et de l'ombre	11/02/2024	14/02/2024
Implémentation des intéractions de blocs	12/02/2024	14/02/2024
Implémentation des collisions	12/02/2024	14/02/2024
Ajout des textures	12/02/2024	13/02/2024
Implémentation des actions nunchuk	13/02/2024	14/02/2024
Implémenation de l'occlusion ambiante	14/02/2024	19/02/2024
Ajout de biomes	14/02/2024	20/02/2024
Implémenation de la gravité	15/02/2024	16/02/2024
Fonctionnalités qualité de vie (locks, collisions v2, sneak)	15/02/2024	20/02/2024
Ajout GUI	19/02/2024	22/02/2024
Développement de la mise en cache	19/02/2024	23/02/2024
Ajout de structures	21/02/2024	23/02/2024
Ajout du mode créatif	20/02/2024	21/02/2024
Ajout de la musique	20/02/2024	22/02/2024
Implémentation du craft	20/02/2024	22/02/2024
Ajout du mode "Survie"	22/02/2024	23/02/2024
Ajouts qualité de vie de l'interface	23/02/2024	23/02/2024

Figure 47 : Diagramme de Gantt réel

Cela implique donc également le fait de devoir répartir les tâches différemment. Voici donc la façon dont nous avons réparti les ressources durant ce projet. Cela est uniquement à titre indicatif car il est très difficile d'estimer la valeur d'une tâche. Ces tâches n'incluent pas non plus toute la recherche de documentation ainsi que la rétro-ingénierie sur le peu d'exemple de démonstrations disponibles sur les outils que nous avons utilisé. Nous sommes très satisfaits de la façon dont nous avons travaillé en équipe. Les passages à plus de 100 % ne signifient pas spécialement que la personne a plus travaillé, mais plutôt qu'elle a dépassé les attentes qu'elle s'était fixé sur cette période. Enfin, nous avons travaillé majoritairement sur l'émulateur Dolphin présenté précédemment, mais à partir du vendredi 9, nous avons pu tester quotidiennement notre avancée sur la console afin d'éviter les mauvaises surprises de performance (Il est aussi arrivé que l'exécutable se lance sur l'émulateur et pas sur la console).

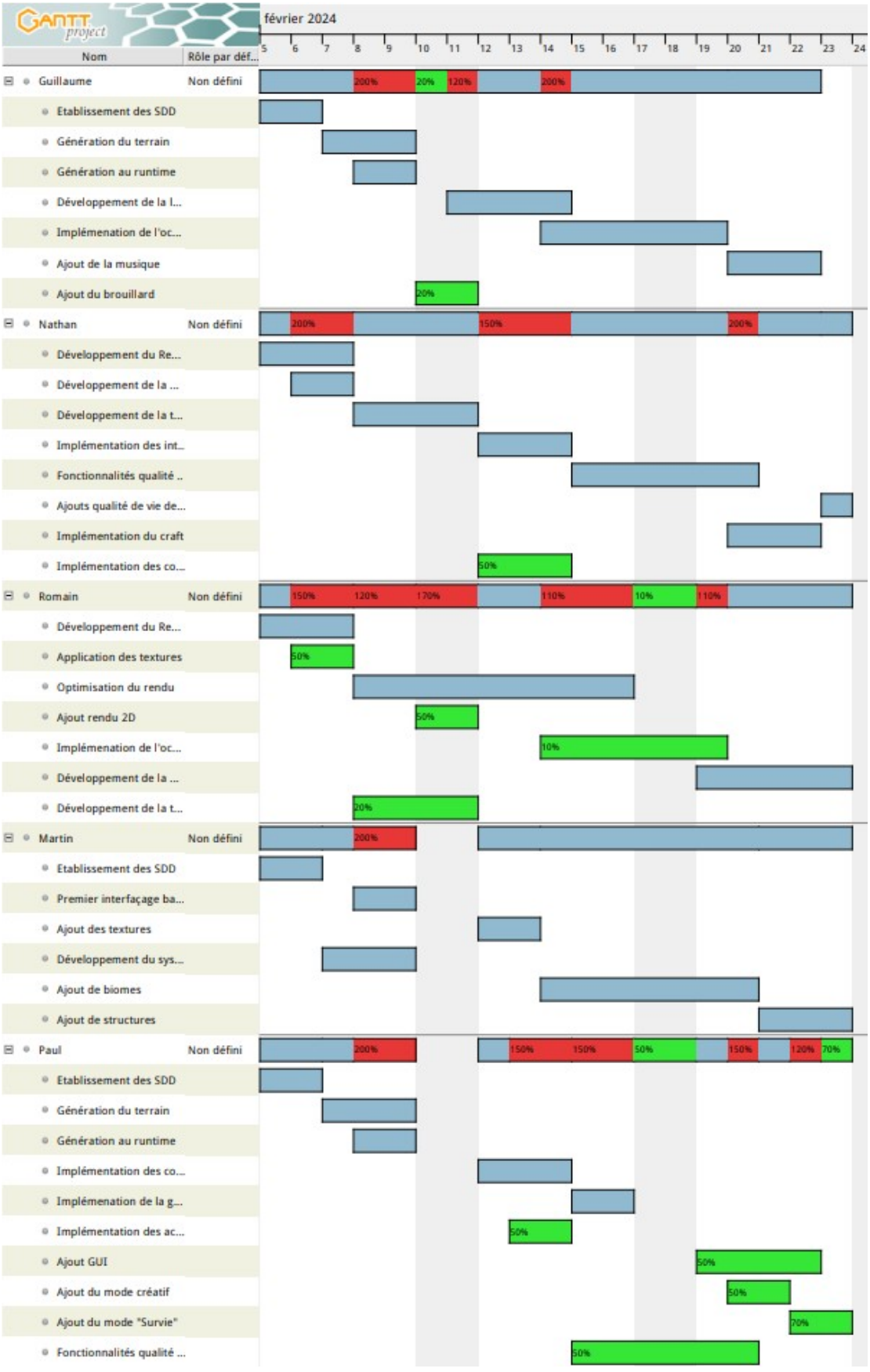


Figure 48 : Répartition indicative des tâches

Présentation du Résultat

Le but de cette section est de faire un tour rapide de l'ensemble des fonctionnalités disponibles dans la version du jeu au moment de la soutenance. Cependant, nous sommes tous motivés à faire avancer ce projet sur notre temps libre afin de lui donner une forme encore plus aboutie.

Le code source du jeu est disponible sous licence MIT sur sa page Git Hub que voici :

https://github.com/MissingNo42/MiiCraft

Fonctionnalités

Génération infinie

Une des fonctionnalités principales de notre jeu est la possibilité d'explorer un monde presque infini et cohérent. Cela est possible avec le système de chargement et déchargements des chunks sur la mémoire vive de la console que l'on exploitera à son maximum. De plus, cette génération n'est pas lassante grâce à une grande diversité de biomes et des reliefs parfaitement inégaux grâce à la génération par bruit détaillée précédemment. Enfin, cette génération restera cohérente même en cas de téléportation du joueur grâce aux propriétés des bruits de gradient.

Lumières

Tous nos travaux sur la lumière nous ont permis d'avoir les fonctionnalités suivantes dans notre version de Minecraft :

- Les ombres portées des blocs et des structures par la lumière du soleil.
- La gestion des sources lumineuses naturelles et artificielles locales
- L'occlusion ambiante

Tous ces ajouts apportent un gros plus qui, même s'il est discret à première vue, donne un aspect beaucoup plus abouti qui manque si on l'enlève à nouveau. Certaines de ces caractéristiques n'étaient même pas présente dans les toutes premières versions du Minecraft original.

Physique

Notre version donne une version de la physique la plus proche possible du jeu de base. Cela inclut des collisions facilitant les déplacements dans les coins exiguës, la gravité correspondant à celle de Minecraft ainsi que la physique de l'eau. Le joueur peut ainsi plonger et nager à sa guise. La seule concession qui a été faite est celle de la nage à la surface. En effet, nous avons choisi d'augmenter volontairement le saut depuis la surface de l'eau afin de faciliter la sortie des lacs et des mers pour le joueur. Il est évidemment possible de désactiver la gravité et les collisions indépendamment pour explorer le monde ou dans notre cas, déboguer.

Commandes

Déplacements

- Pour se déplacer dans le jeu, le joueur utilise le joystick du Nunchuck. Ce contrôle permet une navigation fluide à travers l'environnement, offrant une expérience immersive.

- La caméra est contrôlée à l'aide du pointeur de la Wiimote, permettant au joueur de regarder autour de lui avec précision et facilité.

- Pour sauter, le joueur appuie sur le bouton A de la Wiimote, déclenchant une impulsion verticale pour franchir les obstacles ou explorer les hauteurs.

- Le sprint est activé en appuyant sur le bouton C du Nunchuck, permettant au personnage de se déplacer plus rapidement pour explorer de vastes terrains.

- Enfin, pour s'accroupir, le joueur utilise la flèche de la Wiimote, lui permettant de ne pas tomber.

Interactions avec l'environnement

- Pour poser un bloc dans le monde, le joueur appuie sur le bouton B.

- Le bouton Z du Nunchuck permet au joueur de se concentrer sur un bloc spécifique, ajustant automatiquement la caméra pour focaliser son attention sur cet élément.

- Pour détruire un bloc et le récupérer dans l'inventaire, le joueur effectue un secouement du Nunchuck, déclenchant une action de destruction pour collecter des ressources ou dégager le chemin.

Inventaires et Crafts

- Le déplacement dans la hotbar est réalisé à l'aide des flèches gauche et droite de la Wiimote, permettant au joueur de sélectionner rapidement le bloc souhaité pour une utilisation immédiate.

- Pour prendre ou poser un item de l'inventaire, le joueur appuie sur le bouton A, facilitant la gestion des ressources et des objets collectés.

- En appuyant sur le bouton -, le joueur peut diviser un objet en deux parties ou poser un bloc à l'unité, offrant une plus grande précision dans la manipulation des ressources.

- Enfin, le déplacement rapide dans l'inventaire est activé en appuyant sur le bouton +, permettant au joueur de naviguer rapidement entre les différentes sections de l'inventaire et de réaliser des actions telles que la fabrication rapide d'objets.

RSE

De nos jours, la prise en compte des enjeux environnementaux et sociaux est primordiale dans tous les domaines. L'informatique ne fait pas exception, c'est pourquoi la question devait se poser à chaque étape du projet. On peut se demander en quoi la reproduction d'un jeu vidéo peut jouer un rôle dans la préservation de l'environnement et le bien-être d'une société. Pourtant, nous allons voir que cela peut y contribuer à son échelle.

En effet, le choix de la Wii n'est pas anodin. Premièrement, il s'agit tout simplement d'une des consoles les moins polluantes du marché avec une consommation moyenne du CPU est de 4 V pour 1,5 A, soit moins que certaines clé USB. De plus, la consommation moyenne globale est de 45 W [10] contre environ 150 W pour les consoles de jeux concurrentes ou plus récente et plus de 550 Watts en moyenne pour un PC fixe dédié aux jeux ! Cela représente une économie allant de 15 à 60 grammes de CO2 produit par heure [11] . Autrement dit, 185 personnes jouant à notre version de Minecraft plutôt qu'à la version originale sur un PC fixe économiserai la production de CO2 qu'un trajet de 100 kilomètres en Citroën C1 [12] !

Console	Consommation moyenne (Watts)	KWH	KG de CO2 / Heure
Xbox 360	180	0,18	0,054
Xbox One	122,5	0,1225	0,035
Xbox Series X	–	–	0,07 (estimation)
PS3	190	0,19	0,054
PS4	120	0,12	0,034
PS4 Pro	117,5	0,1175	0,033
PS5	–	–	0,022 (estimation)

Figure 49 : Pollution des consoles de jeux

Cette baisse de la consommation se traduit également par des baisses de prix de la facture d'électricité en cas de sessions de jeu prolongé (utilisation éducative de Minecraft). Couplée au bas prix de la console aujourd'hui disponible d'occasion pour moins de 30 €, nous obtenons une unique réponse à deux enjeux majeurs : l'accès facile et peu coûteux à un mastodonte de l'univers vidéoludique tout en réduisant son impact environnemental.

Conclusion

Pour conclure, nous pouvons dire que nous sommes fières d'avoir atteint et même dépassé nos objectifs initiaux. De plus, le développement de notre jeu a été fait en tenant compte des futurs ajouts possibles. C'est pourquoi nous sommes motivés à poursuivre son développement après la soutenance. Il est par exemple prévu d'ajouter à notre monde la possibilité d'utiliser d'autres blocs fonctionnels, ajouter des outils ou encore ajouter des blocs non-cubiques et des décors. Les possibilités pour la suite du développement sont encore infinies.

Même dans son état actuel, le jeu est très proche du jeu original au moment de sa sortie que ce soit visuellement ou dans les interactions du joueur avec le monde, et cela en sachant que nous étions cinq et que nous ne disposions que de trois semaines. Nous avons su dépasser les contraintes techniques imposées par le support pour fournir un résultat conforme à l'objectif.

Tables des illustrations

Figure 1 : Console Wii noire & Wiimote....5
Figure 2 : Carte des chunks....8
Figure 3 : Hotbar....9
Figure 4 : Inventaire....9
Figure 5 : Caméra 3D (point A) et cône de projection....10
Figure 6 : Types de projection spatiale [5]....11
Figure 7 : Schéma du TEV & formule de calcul [6]....12
Figure 8 : Configuration du TEV en texturing modulated [6]....12
Figure 9 : Tests de modulation et PoC sur la texture de pierre....13
Figure 10 : Espèce de guirlande la plus commune (~98% des rencontres)....13
Figure 11 : R&D sur la transparence....14
Figure 12 : Test des ombres dynamiques GX....15
Figure 13 : Test de lumière de type spotlight....15
Figure 14 : Test des lumières sans textures....15
Figure 15 : Le brouillard comme outil d'occlusion....15
Figure 16 : Un bloc....16
Figure 17 : Le même bloc....16
Figure 18 : Preuve de concept du TexEnv....17
Figure 19 : Nuancier des 16 niveaux d'ombre....17
Figure 20 : Rendu sans occlusion ambiante....17
Figure 21 : Rendu avec occlusion ambiante....17
Figure 22 : Patrons d'occlusion ambiante....17
Figure 23 : Attribution des couleurs aux vertex d'une face....18
Figure 24 : Face d'occlusion après lissage....18
Figure 25 : Preuve de concept de l'occlusion ambiante : seuls les deux vertex supérieurs ont été noircis....18
Figure 26 : cylindre sans (à gauche) et avec (à droite) culling [7]....19
Figure 27 : exemple sans filtrage (à gauche) et avec (à droite) en 2D....19
Figure 28 : surface d'un chunk rendu avec l'adjacence culling....20
Figure 29 : chunks visible en bleu dans le cône de vision....21
Figure 30 : exemple minimal de ce que permet le cache....22
Figure 31 : PoC du mode 2-joueur en splitscreen....22
Figure 32 : Différentes couches de bruits....23
Figure 33 : Biome Savane....24
Figure 34 : Biomes Taïga et Toundra....24
Figure 35 : Montagnes et Conifères....25
Figure 36 : Forêt de cerisiers....25
Figure 37 : Forêt sombre....25
Figure 38 : Carrière de gravier....25
Figure 39 : Une mesa boisée....25

Figure 40 : Couches de générations....26
Figure 41 : Exemple collisions....27
Figure 42 : Exemple de bloc visé....28
Figure 43 : Table de fabrication....28
Figure 44 : Liste des tâches prévisionnelles....29
Figure 45 : Diagramme de Gantt prévisionnel....29
Figure 46 : Attribution prévisionnelle des tâches....30
Figure 47 : Diagramme de Gantt réel....31
Figure 48 : Répartition indicative des tâches....32
Figure 49 : Pollution des consoles de jeux....35

Lexique de Minecraft

Chunk : Subdivisions du monde en ensemble de 16X16X128 blocs.

Craft : Fabrication de blocs ou objets à travers l'inventaire ou la table de fabrication.

Hotbar : Barre d'accès rapide de l'inventaire accessible depuis tous les inventaires possibles et depuis laquelle on peut poser des blocs ou utiliser des objets.

Biomes : Régions du monde aux climats et décors variés.

Bedrock : Couche incassable délimitant les dernières couches de profondeur du monde

Bibliographie

[1] Nintendo, «Dedicated Video Game Sales Units,» 31 12 2023. [En ligne]. Available: https://www.nintendo.co.jp/ir/en/finance/hard_soft/index.html. [Accès le 25 2 2024].

[2] Wikipédia, «Wii,» 10 2 2024. [En ligne]. Available: https://fr.wikipedia.org/wiki/Wii#Sp%C3%A9cifications_techniques. [Accès le 25 2 2024].

[3] Nintendo, «Revolution SDK - Graphics Library (GX),» 27 3 2009. [En ligne]. Available: https://pokeacer.xyz/wii/pdf/GX.pdf. [Accès le 9 2 2024].

[4] Nintendo, «Wii SDK PDF List,» [En ligne]. Available: https://pokeacer.xyz/wii/pdf/. [Accès le 12 2 2024].

[5] Wikipédia, «3D projection,» 28 1 2024. [En ligne]. Available: https://en.wikipedia.org/wiki/3D_projection. [Accès le 26 2 2024].

[6] Nintendo, «Wii Graphics Primer,» 15 2 2008. [En ligne]. Available: https://pokeacer.xyz/wii/pdf/WiiGraphicsPrimer.pdf. [Accès le 9 2 2024].

[7] K. Power, «Backface culling,» 12 12 2012. [En ligne]. Available: https://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/HSR/backfaceculling.html. [Accès le 26 2 2024].

[8] KdotJPG, «NoisePosti.ng,» 16 01 2022. [En ligne]. Available: https://noiseposti.ng/posts/2022-01-16-The-Perlin-Problem-Moving-Past-Square-Noise.html. [Accès le 11 02 2024].

[9] H. Kniberg, «Youtube,» 06 02 2022. [En ligne]. Available: https://youtu.be/CSa5O6knuwI?si=djh23BFjDAZM_HkV. [Accès le 17 02 2024].

[10] «ATI Wii GPU: caractéristiques techniques et tests,» *technical.city,* 2016.

[11] S. Roche, «L'impact environnemental du jeu vidéo,» *playstationinside.fr,* 2022.

[12] «Consommation de la Citroen C1 1.0I 2005 à 2007,» *I-Itineraire.com,* 2015-2024.