PyCraft - Documentation

Oscar Valayé, Stanley Ramanantsoa et Arthur Sirvent

Trophées NSI 2024

Sommaire

1 Guide Utilisateur

1.1 Requirements

- Un ordinateur assez performant (à la fois en termes de CPU, GPU et RAM)
- Python 3.10 ou 3.11
- Une possibilité d'écriture disque
- Un espace disque d'au moins 200 Mb pour le monde
- Les librairies numpy, pygame, moderngl, pynput, pywin32, (voir le README pour plus de details)

1.2 Procédure d'utilisation

Lire le README.md

1.3 Reboot

Pour supprimer tous les mondes que vous avez créé, executez le fichier "reboot.bat" (qui execute reboot.py). Si il ne marche pas, executez le fichier "reboot.py", de la meme manière que vous executez le fichier "main.py" pour lancer le jeu.
On vous demandera alors de confirmer, pressez "o", et la suppression commencera. Tous les dossiers de mondes existants ainsi que les données du monde seront supprimés.

1.4 Contrôles du jeu

Vous pouvez aller dans preferences.txt pour modifier les préferences d'utilisation (dont un paramètre clavier), mais à vos risques et perils ;)
Voici les contrôles de base, qui sont très similaires au Minecraft de base.
- Echap : met le jeu en pause, libère la souris (dans le jeu elle est bloquée et invisible).
- A + Echap ou Croix de la fenêtre : quitte le jeu
- Z, S, Q, D : Avancer, reculer, gauche, droite
- Espace : Sauter (au sol) ou monter (lors du vol)
- Shift-Maj : Descendre (lors du vol)
- Mouvement de la souris : changer l'orientation
- Clic droit : placer un bloc
- Clic gauche : casser un bloc
- F : changer de mode de déplacement (de vol à normal et de normal à vol)
- Fleches droite/gauche : changer d'item dans la barre d'inventaire.
- Fleches haut/bas : Changer de barre d'inventaire

1.5 Problèmes possibles

Il est éventuellement possible de des problèmes surviennent lors de l'execution. Il y a 2 categories : Les erreurs qui arrêtent le programme, et les bugs d'affichage, qui eux ne font pas d'erreurs.

1.5.1 Erreurs

Il peut en effet y avoir des erreurs, mais c'est très rare, que des erreurs surviennent. En voici quelques unes :
1. BrokenPipeError : cette erreur n'est pas de notre faute. C'est une erreur système où le Pipe entre les data manager est mal initialisé. Attendez un peu et relancez le jeu.
2. allow-pickle == False / EOFError / No data left in file : Cette erreur est très rare, elle arrive si jamais le jeu a été précedemment quitté au milieu d'une sauvegarde numpy, qui corromp temporairement le fichier. Il suffit juste de relancer pour que ça marche.
3. MemoryError : La RAM disponible était insuffisante, l'erreur a donc été déclenchée. Cela peut être à cause d'un programme qui consomme beaucoup de RAM de manière exceptionnelle, lors de l'ouverture par exemple. Il suffit dans ce cas de relancer. Cela peut aussi être du à une quantité de RAM insuffisante sur l'ordinateur, dans ce cas changez de machine.
4. OSError, numpy array n'a pas pu être écrit : Cela veut dire qu'il n'y a plus de place sur l'ordinateur. Faites de la place puis relancez.
5. concatenation error : cette erreur est bénine, elle est due à une erreur précedemment dans l'execution.

1.5.2 Bugs d'affichage

Il peut y avoir plusieurs bugs d'affichages, la plupart assez bénins :
1. Triangle et bandes de couleurs : ce bug d'affichage n'arrive normalement pas sur un ordinateur normal. Dans ce cas, c'est la faute de vos drivers qui peuvent être corrompus ou non adaptés. Mettez à jour vos drivers, ou changez d'ordinateur si le bug persiste et qu'il vous gêne vraiment (en effet, le bug peut être plus ou moins genant, allant de simples petites colonnes de couleur à un affichage completement bugué, avec des lignes de couleurs parfois tremblottantes et une fenetre bariolée)
2. Blocs fantômes : si il y a des blocs au travers desquels vous pouvez passer à travers, que vous ne pouvez pas casser, et qui ne semblent pas à leur place. C'est dans ce cas probabalement la faute d'une corruption mémoire, ou d'une mauvaise initialisation de la mémoire avec vos drivers.
3. Chunks absentes non affichées au milieu de chunks affichées, qui ne s'affichent toujours pas au bout d'un certain temps : Le bug n'arrive normalement plus, si il arrive, c'est dû à une mauvaise gestion des structures, avec une chunk qui est oubliée. Si cela arrive, il faut partir loin de la chunk buguée (pour qu'elle soit unload), puis revenir. Quand elle sera load à nouveau, elle sera affichée. Vous pouvez aussi quitter le jeu et load le monde pour que ça marche.
4. Grands rectangles de lave en dehors d'un volcan : on ne peut rien y faire, cela arrive si il y a plusieurs volcans juste à côté.
Veuillez noter que si les bords du volcan sont génerés avant son centre (là où il y a la lave), ce n'est pas un bug, c'est nécessaire pour placer la lave à la bonne hauteur (voir la partie correspondante dans volcano.py).

1.5.3 Problèmes de performance

Si vous avez un problème de performances et que le jeu est trop lent (en particulier pour la generation de terrain), voici les solutions possibles :
- Supprimer ou diminuer le nombre de structures (voir dans preferences.txt), voire desactiver les îles flottantes.
- Diminuer la gen-distance et la render-distancee (voir preferences.txt)
Veuillez noter dans la plupart des cas, si il y a des problèmes, cela est dû à un CPU/GPU pas assez performant, pas assez de RAM ou à une écriture disque trop lente (si on est sur une clé USB ou en serveur par exemple).

2 Introduction générale

2.1 Notre jeu

Nous avons codé PyCraft depuis septembre, à partir de zéro (c'était notre objectif) sans grosse librairie qui fasse tout à notre place. Le but était de recréer Minecraft, mais avec certaines additions (par exemple des volcans, des îles flottantes ou une gravité dynamique) et en python, ce qui allait poser des contraintes de performance. Nous voulions également que notre jeu soit capable de tourner de façon satisfaisante sur les ordinateurs de notre lycée qui ne sont pas les plus performants.
Nous avons, comme Minecraft, le déplacement dans un monde infini, le placement et le cassage de blocs, les biomes, les arbres, les villages etc.

2.1.1 Les 4 grandes parties

Il y a 4 grandes parties à ce projet et cette section ne servira que de l'introduction à chacune d'entre elles, afin d'avoir une idée générale de la structure du jeu.
Les parties sont ensuite explicitées en grand détail dans des sections en dessous.
Les 3 parties sont donc la génération de terrain, l'affichage et le jeu. Nous verrons quelle est la 4ème après.

La génération de terrain
La première partie est la génération de terrain qui a été entièrement gérée par Arthur. Cette partie s'occupe de générer le terrain du jeu, *intégralement*, partant du relief avec les montagnes, les océans, les déserts et les plaines, jusqu'aux villages, volcans et îles flottantes tout en passant par des arbres et des minerais.
Cette partie est très chronophage, et nécéssite donc (pendant que le jeu tourne, le début est un peu plus complexe) d'être séparée dans un autre processus. Afin de toutefois toujours pouvoir communiquer avec le processus du jeu, la génération de terrain utilise les data manager, une paire de fichiers assurant la connection entre les processus du jeu.
Tous les fichiers de génération peuvent être trouvés dans le dossier terrain_generation.

L'affichage du jeu
La seconde grande partie du jeu est l'affichage de celui-ci. Elle est maintenant gérée par Stanley, et l'a été petit peu par Oscar.
Étant donnée la nature en python du jeu, et le coût de projections 3D (2 matrices de rotation 3x3 et 1 matrice de projection *par point*), effectuer l'affichage avec un module standard python uniquement (tkinter ou pygame) est hors de question.
Ainsi, Stanley s'est tourné vers le GPU qui est capable d'effectuer ces opérations d'affichage 3D en parrallèle (tout en même temps), en opposition au CPU, qui projette les points un par un. De plus, le GPU est construit spécifiquement pour ces calculs.
Le seul problème qui soit apparu est celui de la communication avec le GPU et ses programmes: les shaders. En effet, celle-ci ne se fait qu'en utilisant un language, le GLSL. Cependant, pour rester dans les règles du concours, nous avons donc utilisé une librairie pour précompiler du python en GLSL, de telle

sorte à ce que malgré le fait que du GLSL soit présent dans le code du projet, il n'a pas été codé par nous, mais traduit du python par une librairie(nous avons cependant dû modifier le résultat compilé pour que ce dernier fonctionne-c'est tout de même mieux) Vous pouvez trouver ces shaders à game/shaders/shaders.py). Nous avons également fait de notre mieux pour restreindre le plus possible la quantité de glsl dans le jeu (nous ne sommes qu'à 130 lignes de glsl pour tout ce dernier)
Tous les codes de l'affichage se trouvent donc dans la section rendering ainsi que shaders.

Le Jeu
Nous avons à présent parlé de comment générer le terrain du jeu ainsi que de comment afficher ce dernier, mais nous n'avons pas encore parlé du jeu en lui même. C'est sur cela que porte la 3ème partie de cette documentation, soit le lien entre génération et affichage (les data manager), en passant par les méchaniques du jeu telles que le fait de pouvoir bouger, se tourner, casser et placer des blocs, et toute la simulation de la physique. C'est également ici que se trouvent les matrices de rotation et de projection.
Cette partie a été essentiellement gérée par Oscar, essentiellement, puisque le grand rôle de cette partie étant de lier les autres, une grande communication et collaboration constante a été requise pour effectuer des mises à jour de parties (que nous avons appellées merges), qui pouvaient durer des semaines à cause de bugs

Fichiers autres
Il y a également une catégorie de fichiers rentrant techniquement dans la partie jeu, même si elle est un peu générale. On y trouve les fichiers
-main.py, soit le fichier maître appellant tout - Codé par Arthur
-game_manager.py, soit le fichier censé gérer le début et la fin d'une session de jeu - Codé par Arthur
-settings.py, soit un fichier dans lequel sont définies une multitude de constantes utilisées un peu partout - Codé par tout le monde
-préférences.py, soit le fichier récupérant les préférences du fichier preferences.txt, soit l'équivalent des settings, mais censé être changeable par l'utilisateur (paramètres clavier par exemple) - Codé par Oscar

La documentation
Et oui! La dernière partie de notre jeu est sa documentation extensive et exhaustive, concue dans le but que n'importe muni de notre code ainsi que de cette documentation, soit capable de comprendre pourquoi et comment ce projet de 8 mois a pu être développé, et pour ainsi plus facilement pouvoir améliorer ce dernier.
Nous avons mis cette étape comme une partie, puisque ça nous a probablement pris plus de 50 heures combinées de construire cette documentation de la façon la plus complète possible, et en LaTeX.
Nous éspérons véritablement que cette ressource pourra venir en aide à d'autres personnes ayant en tête un projet similaire au notre.

2.2 Concepts partagés entre les différentes parties du jeu

Ici nous allons voir quelques concepts qui sont utilisés dans les 3 parties sans pour autant y être définis. Nous allons donc commencer avec

2.2.1 Identifier le type de bloc

Nous avons plusieurs blocs dans le jeu. Afin de pouvoir facilement les identifier avec aussi peu d'espace

que possible, nous avons donc un identifiant (un int) par bloc, le représentant. Nous avons donc deux dictionnaires dans settings.py, ID_VERS_BLOC ET BLOC_VERS_ID.
Il y a également les dictionnaire id_vers_tex et tex_vers_id qui représentent eux la conversion id de bloc à id de texture, utilisé pour l'affichage.

2.2.2 Le découpage du terrain: les chunks

Une notion centrale à la façon de laquelle le jeu est géré est celle d'une chunk, qui est en français traduite par "tronçon" ce qui n'aide pas franchement. Essentiellement, une chunk est dans notre jeu une subdivision du terrain en grands carrés de 16x16x300 blocs, avec les 16x16 étant les dimensions horizontales, et le 300 la dimension verticale. Cela nous permet donc de codifer quelles parties du terrain nous cherchons à générer ou à afficher. Cela simplifie également énormément le processus du stockage du monde sur disque. Voici une image du jeu dans laquelle nous avons illustré les chunks:

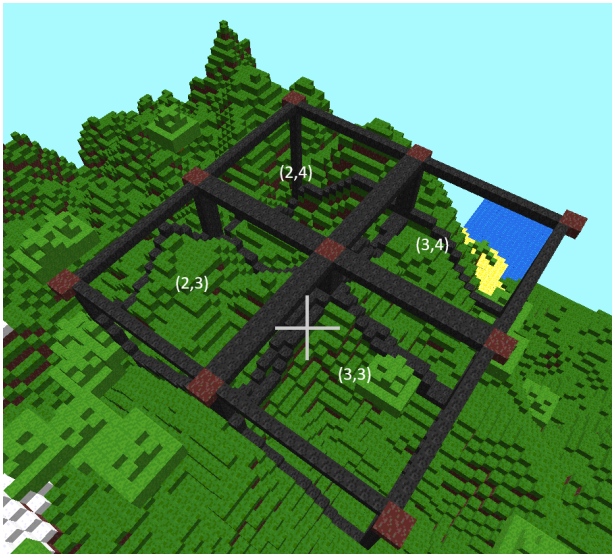


Figure 1: Illustration des chunks: représentés par des blocs de basalte, avec des blocs de terre pour les endroits où il se rencontrent

Pour identifier les chunks, nous leur associons des coordonnées sous format (chunk_x, chunk_y), avec un changement de chunk correspondant à une paire de coordonnées différant de 1 dans les coordonnées (nous en avons mis des arbitraires sur le schéma pour illustrer).
Ces coordonnées sont alors dénommées dans les programmes et dans la doc comme *coordonnées chunk*, *coordonnées normalisées* ou encore *coordonnées normalisés chunk*

2.2.3 Éviter l'affichage inutile: les renderables

Lors de l'affichage des blocs d'un chunk, que quelques-uns sont visibles, principalement ceux en surface (on ne voit par exemple quasiment jamais de la pierre en surface, et pourtant c'est un des blocs les plus abondants). Ainsi, ce serait se tirer une balle dans le pied que d'essayer d'afficher tous les blocs dans une

chunk.
Donc, pour rémédier à ce problème, nous avons mis en place les blocs "renderables", ou les "renderables" d'une chunk, qui sont justement la liste de tous les blocs *visibles*.
Voici un exemple affiché de renderables d'une chunk (ce qui est véritablement affiché dans le jeu)

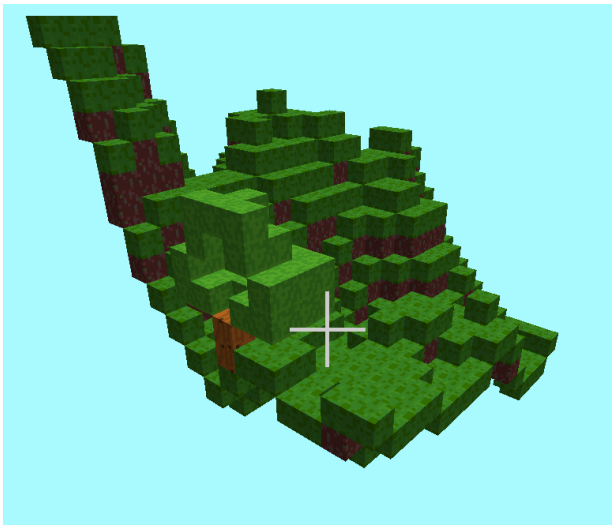


Figure 2: Les renderables d'une chunk, soit les blocs visibles

NB: le fait d'avoir des renderables complique grandement le cassage ainsi que le placement de blocs, puisqu'il faut alors les mettre à jour et ajouter des blocs aux blocs affichés.

2.2.4 La génération de structures anticipée: les structure scan

Vous avez sûrement remarqué les structures présentes dans le jeu (volcan et village). Afin que celles-ci se génèrent, il faut effectuer un "scan" sur une grande zone (un *overchunk*, de 512x512 blocs).
Pourquoi?
Assez simplement, pour qu'une structure se génère il faut que son centre ait été récelé. Or, si lorsque cela se produit, les chunks environnants ont déjà été générés, nous ne pouvons pas générer la structure (imaginez que vous avanciez dans le jeu, et que soudain un volcan apparaisse sous vos pieds). Donc, sans structure scan pour pouvoir prévoir les structures, celles-ci ne pourraient tout simplement pas exister!

2.2.5 Choix d'optimisation globaux

Nous avons tout au long fait plusieurs choix, afin d'optimiser le jeu, particulièrement au niveau des variables en python. Nous avons donc utilisé des
-np(=numpy).array Les arrays de numpy étant codées en C leur permettent d'être nettement plus performantes sur une multitude de niveaux quand comparées aux listes python. C'est sous ce format que les chunks et donc les mondes sont stockés(.npy est nettement plus compact, rapide et pratique que .json).
-set() En effet, les ensembles python peuvent paraître inutiles, puisque l'ordre n'est pas préservé, toutefois, cela vient avec de nombreux avantages: l'opération "element in set()" est O(1) contrairement à l'opération

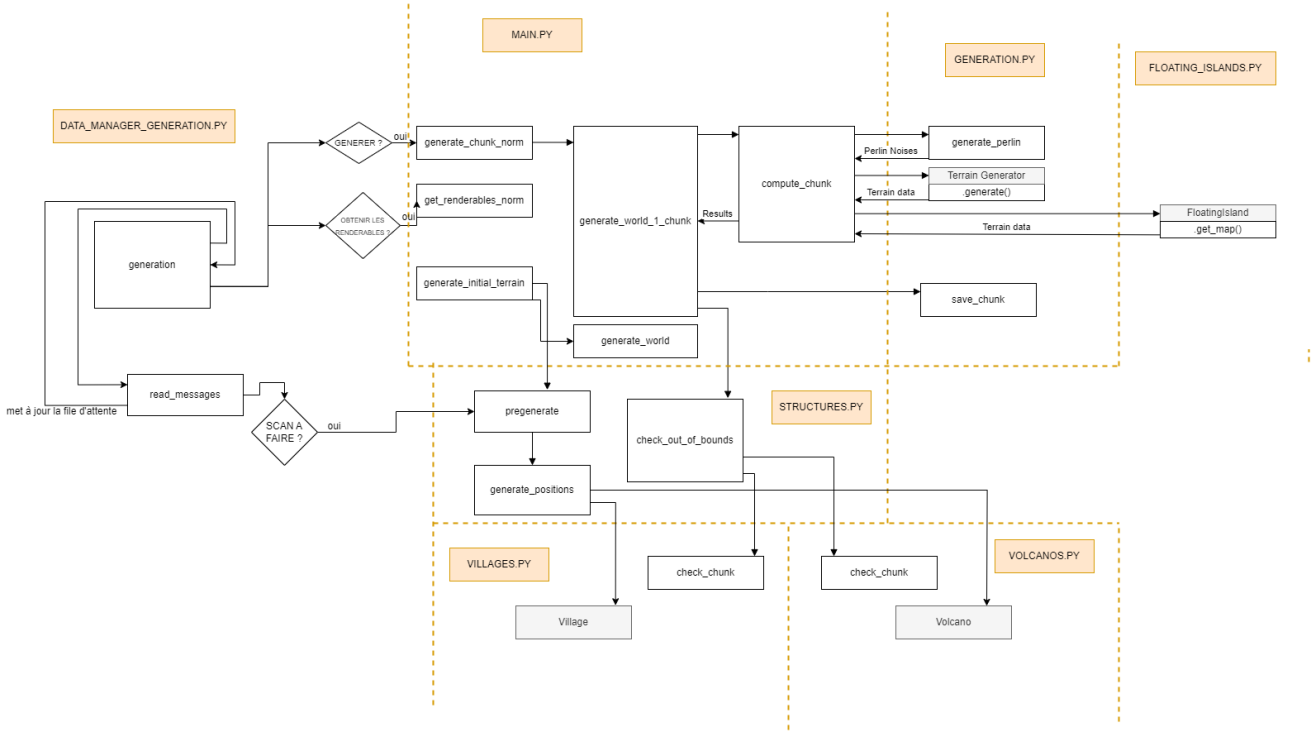
"element in list()" qui elle est O(n) avec n la longueur de la liste. Cela fait donc que pour des variables telles que les chunks affichés, ou les chunks générés, un ensemble est la solution idéale. De plus, avec ces-derniers, nous évadons complêtement le potentiel problème que pourrait poser un duplicat (blocs dans une chunk).
Voici donc pourquoi vous verrez beaucoup d'ensembles et de np.array dans nos fichiers.

3 La Generation de Terrain

Introduction

3.1 Architecture génerale

Figure 3: Architecture génerale de la partie géneration de terrain



Les fonctions principales de géneration et de renderables sont appellées par le data_manager_generation.py ou par le game_manager.py pour la géneration intiale de terrain

3.2 Algorithmes

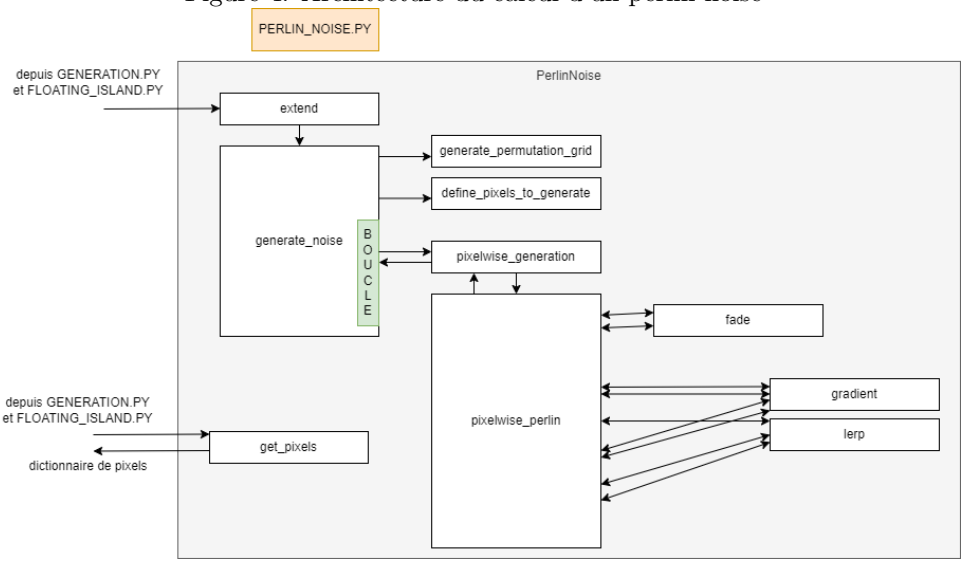
3.2.1 Perlin Noise

Le Perlin Noise, ou bruit de perlin, est l'algorithme à la base de la géneration procédurale de terrain. Il permet de génerer une zone 2D avec des variations harmonieuses.

Paramètres

Il prend comme paramètres : une amplitude, un nombre d' octaves et une persistence.
L'amplitude determinera simplement l'échelle du bruit, c'est à dire si le perlin noise est agrandi dans l'espace 2D.
Le nombre d'octaves est une valeur qui déterminera le nombre de couches que comprend le perlin noise. Par exemple, une valeur de 3 signifie qu'on va génerer 3 couches, chacune ayant une fréquence deux fois plus petite que la précédente.
La persistence définit à quel point l'amplitude change en fonction des couches d'octaves. Une persistence de 1 signifie que chaque couche aura le même poids, tandis qu'une persistence de 0.5 signifie que la couche suivante aura un poids 2x inférieur à celle d'avant (la plus grande).

Figure 4: Architecture du calcul d'un perlin noise



perlin_noise.py Classe PerlinNoise()

__init__ (self, SEED, noise_scale, octaves, noise_persistence, artifical_frequencies =None)
SEED : str, seed du perlin noise (qui détermine la grille de permutations)
noise-scale : int, amplitude du perlin noise
octaves : int, nombre d'octaves à generer
noise-persistence : int, persistence du perlin noise
artifical-frequencies : list, fréquences artificielles, generées en plus des octaves normales.

extend(self, top_left, bottom_right)
top-left : tuple, coin supérieur gauche du rectangle à generer
bottom-right : tuple, coin inférieur droit du rectangle à generer
Fonction appellée par main.py et par floating_island .py, Appelle la generation de perlin noise du top-left jusqu'au bottom-right, ajoute la zone au perlin noise déjà géneré si un perlin noise a déjà été généré

generate_noise(self, top_left, bottom_right)
Appellée par self.extend(), cette fonction genere le Perlin Noise entre le top-left et le bottom-right.

D'abord, appelle self .generate_permutation_grid(), puis déinis les pixels à génerer en appellant self . define_pixels_to_genera
Fais une boucle x et y pour quadriller la zone à génerer, puis génère le perlin noise pour chaque pixel et pour chaque fréqence, en appellant la fonction self . pixelwise_generation ()

generate_permutation_grid(self)
Initie la grille de permutation, responsable de l'aléatoire dans le Perlin Noise, et controllé par une SEED spécifique. On a une liste de taille 255*2, mélangée aléatoirement.

define_pixels_to_generate (self, top_left, bottom_right)
Definis les valeurs à génerer, entre le top-left et le bottom-right, qui creée une liste de x-pixels et y-pixels à génerer, puis calcule la longueur et largeur de la zone à genérer.

pixelwise_generation(self, x_pixel, y_pixel, frequency, weight)
(x-pixel, y-pixel) : (int, int) , coordonées xy du bloc
frequency : float, fréquence de géneration
weight : int, poids de l'octave, à quel point il va être important dans le perlin noise final
Retravaille les coordonnées données en fonction des paramètres du Perlin Noise (amplitude, fréquence et poids), puis envoie ces coordonnées à self . pixelwise_perlin

pixelwise_perlin (self, x, y)
(x, y) : (float, float), coordonnées retravaillées
La fonction genere les valeurs du perlin noise pour 1 pixel.

Definit d'abord les coordonnées absolues et les coordonnées relatives dans la cellule, puis applique une fonction de lissage de cette valeur. Ensuite, on calcule les angles des quatre sommets de la cellule, puis on applique un gradient à chacun des sommets. On interpole ensuite linéairement horizontalement, puis verficalement. Retourne cette valeur.

get_pixels (self)
Renvoie tous les pixels (et leur valeur) générés précedemment avec self.extend() sous format de dictionnaire

3.2.2 Renderables

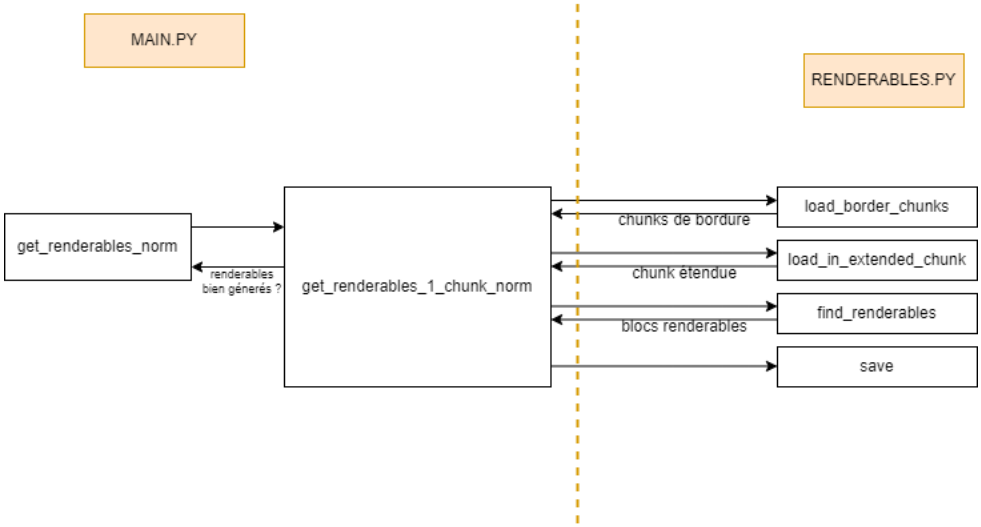
EDITO : "renderables" est un mot que nous avons "inventé" qui vient de l'anglais 'render', et qui signifie "les blocs pouvant être affichés à l'écran".
En effet, par souci d'optimisation, nous n' allons pas envoyer TOUS les blocs du jeu à être affichés, ce serait inutile, puisque on ne va par exemple dans tous les cas ne jamais voir un bloc en profondeur.
Les renderables sont donc tous les blocs qui sont adjacent à un bloc d'air (c'est un peu plus compliqué pour l'eau), et qui seront potentiellement vus par le joueur. Pour chaque bloc de terrain, on va donc verifier les 6 blocs autour de lui (+x -x +y -y +z -z). Si un de ces blocs est de l'air ou de l'eau, le bloc pourra donc être vu, il est donc ajouté aux renderables. Les autres quant à eux ne seront pas renderables.

Voici une explication visuelle en 2D (en vrai c'est en 3D) pour simplifier, mais c'est la même idée en 3D :

Figure 5: Fonctionnement 2D des renderables

A gauche, nous avons les blocs de terrain. A droite, les "renderables". Les blocs verts seront classifiés comme "renderables", ils pourront être vus. Cependant, les blocs rouges du milieu ne le seront pas, ils ne pourront dans tous les cas pas être vus.

Figure 6: Architecture de l'obtention des renderables



renderables.py

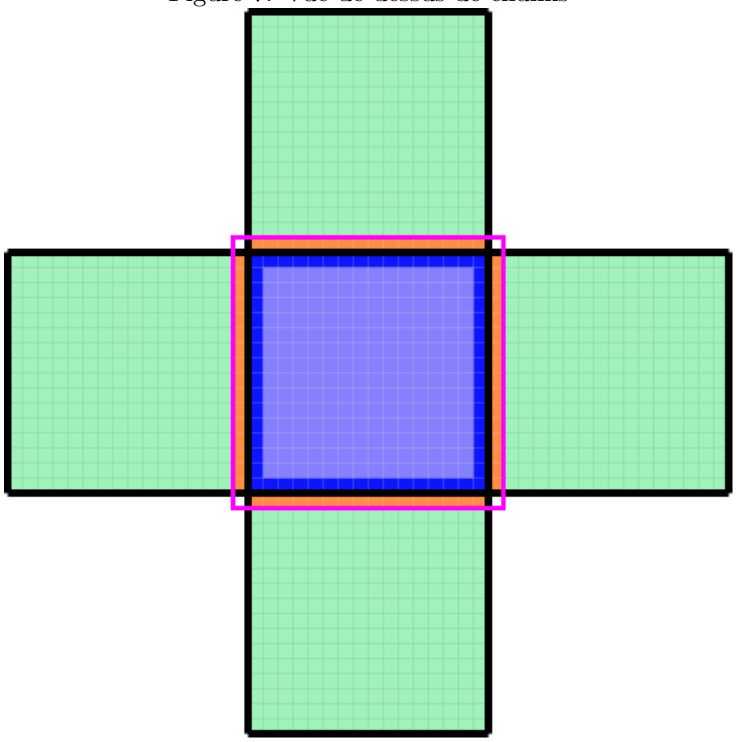
load_border_chunks(top_left)
Load les chunks qui sont en adjacents à la chunk principale pour pouvoir bien génerer les renderables. En effet, pour déterminer si un bloc en bordure de chunk est renderable, on a besoin de la chunk à côté. Par exemple, un bloc en (0, 8, 100) est tout à gauche de la chunk 0, 0. Le bloc à gauche de ce bloc est donc dans la chunk d'à côté, ce bloc est donc le bloc tout à droite de la chunk juste à gauche (ici la chunk -1, 0).

Cette fonction va donc verifier si tous les chunks adjacents existent, en les chargeant. Si ce n'est pas le cas, les renderables ne peuvent pas être déterminés pour cette chunk.

load_in_extended_chunk(extended_chunk, border_chunks_coords)
extended-chunk : np array, chunk dont la taille est étendue de 1 dans les directions +x -x +y -y +z -z pour y mettre les bordures
border-chunks-coords : dictionnaire de np arrays, chunks adjacents
Load les bordures des chunks adjacents dans un grand array principal étendu. Ce grand chunk étendu comprendra donc toute la chunk + les bodures des chunks adjacentes et qui sera de taille 18 x 18 x (MAX-HEIGHT + 2)

Voici un schema qui explique mieux la situation :

Figure 7: Vue de dessus de chunks



Ici, les blocs bleus (foncé+clairs) sont la chunk de base dont on veut obtenir les renderables. Sauf que pour obtenir les blocs bleus foncé, il nous faut les blocs oranges, qui sont situés sur une autre chunk. On

va donc load les 4 autres chunks vertes autour. Le carré violte correspond à l'extended_chunk / grand array

find_renderables(chunk_array)
chunk-array : np array, étendu (voir ci-dessus), qui contient la chunk principale et les bordures des chunks adjacents
Algorithme principal d'obtention des renderables dans un array étendu.

L'algorithme de base est très simple, on itère sur x, y, z pour un range(1, 18) pour les axes x et y (de taille 18), on élimine la première et dernière valeurs qui ont été ajoutées. On itère z sur range(1, MAX-HEIGHT +2). Pour chaque itération, on check les 6 blocs en +x -x +y -y +z -z. La valeur de l'air est 0, celle de l'eau est 5. Si le bloc n'est pas de l'eau, si leur multiplication est un multiple de 5 (donc si il y a soit de l'eau soit de l'air), on les met dans les renderables. Si le bloc est de l'eau, on regarde si cette multiplication est 5^6, si c'est le cas, tous les blocs qui sont autour sont de l'eau, le bloc n'est donc pas renderable. Dans tous les autres cas, il l'est.

Cependant, cette approche est très peu efficace car elle demande beaucoup de tours de boucle. On utilise donc la puissance de numpy. On va en effet créer 6 autres arrays respectivement décalés de +x -x +y -y +z -z. On va multiplier ces 6 arrays, ce qui nous donnera le même résultat qu'en haut, mais directement dans un grand array. On applique ensuite des verifications d'égalité (comme ci dessus).

save(renderables, coords, path)
renderables : np array, array à sauvegarder
coords : tuple, équivalent à top-left, coordonnées de l'array de renderables généré
path : str, chemin de sauvegarde
Sauvegarde les renderables.

3.3 Explication des fonctions

3.3.1 main.py

FONCTIONS PRINCIPALES :

generate_chunk_norm(chunks: list)
chunks : liste, [[chunk1-x, chunk1-y], [chunk2-x...] sous format normalisé, chunks à génerer
Generation real-time appellée par data_manager_generation.py, génère 1 chunk de 16*16.

Pour chaque [chunk_x, chunk_y], on convertit les coordonnées normalisées en coordonnées de blocs, et on appelle la fonction generate_world_1_chunk().

get_renderables_norm(chunk_x: int, chunk_y: int)
(chunk-x, chunk-y) = (chunk0-x, chunk0-y), coordonnées x,y normalisées d'UNE chunk.
Calcul des 'renderables', appellé par data_manager_generation.py.

On convertit les coordonnées normalisées en coordonnées de blocs, et on appelle la fonction get_renderables_1_chunk()

generate_initial_terrain (map_coords)
map-coords : tuple de tuple, (coin haut gauche, coin bas droit), rectangle à génèrer lors de la géneration intiale

Cette fonction s'occupe de la generation initiale du monde, avec un multiprocessing pour accélerer cette géneration.

Prégénère artificiellement les structures au début sur une zone définie, en appellant structures.pregenerate(). Ensuite on génère le monde initial en appellant la fonction generate_world() qui utilisera du multiprocessing pour accélerer, contrairement à generate_world_1_chunk()

FONCTIONS DE GENERATION :

generate_world_1_chunk() Cette fonction génère le terrain pour 1 chunk.

Calcule d'abord le terrain de la chunk, en appellant compute_chunk() avec les coordonnées top-left bottom-right Ensuite, sauvegarde la chunk calculée en appellant generation.save_chunk(), puis ajoute les structures à cette chunk en appellant structures.check_out_bounds().

generate_world_1_chunk(map_coords)
map-coords : tuple de tuple, (coin haut gauche, coin bas droit), rectangle de la chunk à génerer. C'est globalement similaire à generate_world_1_chunk(), sauf qu'on génère le terrain pour beaucoup de chunks, en utilisant du multiprocessing pour optimiser au maximum. En effet, on va répartir le terrain à diviser en plusieurs processus indépendants et qui vont chacun génerer un bout du terrain, tous en parrallèle. On divise d'abord le terrain à génerer en n différents processes (en fonction du nombre de coeur de l'ordinateur). Ensuite, on crée ces processes sur la fonction compute_chunk(), avec les coordonnées top-left bottom-right que le process gère (défini précedememnt).
Ensuite, quand tous les processes sont finis, on récupère les résultats (stockés dans une variable commune à tous les processes). On rassemble ces résultats dans un grand array avec generation. concatenate_results (), qu'on découpe ensuite en chunks avec generation.convert_into_chunks(). On sauvegarde ensuite les chunks découpées avec generation.save_chunks(), puis on appelle la fonction check_for_structures ().

compute_chunk(top_left, bottom_right, result_dict , process_id, offset_tree_placeholders)
(top-left, bottom-right) : (coin haut gauche, coin bas droit), rectangle de la chunk à calculer
result-dict : dictionnaire (partagé entre les processes en cas de multiprocessing) qui contiendra tous les résultats de la géneration process-id : numéro du process (surtout utile en cas de multiprocess) out-of-bounds-tree-placeholders : dictionnaire partagé entre les processes, qui contient les bouts d'arbres qui dépassent de la zone à generer. C'est la pièce centrale de la géneration. Dans cette fonction on fait tous les calculs de géneration d'un bout du monde , défini par son top-left bottom-right, avec le dictionnaire de résultats qui stocke les résultats génerés.

D'abord, on génère les Perlin Noises avec generation. generate_perlin (). Ensuite, on génère le vrai terrain à l'aide de ces perlin noises, qui sont l'équivalent des paramètres du monde, en créant l'instance generation.TerrainGenerator() et en appellant sa fonction .generate(). Si les îles flottantes sont activées, on ajoute les îles flottantes avec floating_island .FloatingIsland (). get_map(). On retourne ensuite les résultats, ou alors on met à jour le dictionnaire commun à tous les processes avec la data calculée.

compute_mineral_proba()
Précalcule les probabilités de spawn des minéraux pour chaque hauteur z, pour éviter de faire les calculs en real time. Les probabilités de spawn du fer, diamant et charbon sont calculées selon des courbes.

FONCTIONS DE RENDERABLES :

get_renderables_1_chunk(top−left)
top-left : coin haut gauche de la chunk à génerer, = coordonnées de bloc de la chunk Pour 1 chunk donnée et ses coordonnées, cette fonction génère les renderables.

D'abord, verifie que la chunk de base existe, sinon retourne False. Ensuite, appelle la fonction renderables.load_border_chun

Si cette fonction renvoie False, les chunks adjacents n'existent pas tous, retourne False. Crée ensuite le grand array étendu de 1 dans les 6 directions (-x +x -y +y -z +z). Remplis le centre (sans les bordures) du grand array avec la chunk de base (loadé plus tôt). Après, remplis les bordures du grand array avec les chunks adjacentes loadés plus tôt, en appellant renderables.load_in_extended_chunk(). Ensuite, calcule les blocs renderables en appellant renderables.find_renderables(), avec en argument le grand array étendu. Finalement, sauvegarde ces blocs renderables avec renderables.save(), puis retourne True, signifiant que la fonction a bien marché

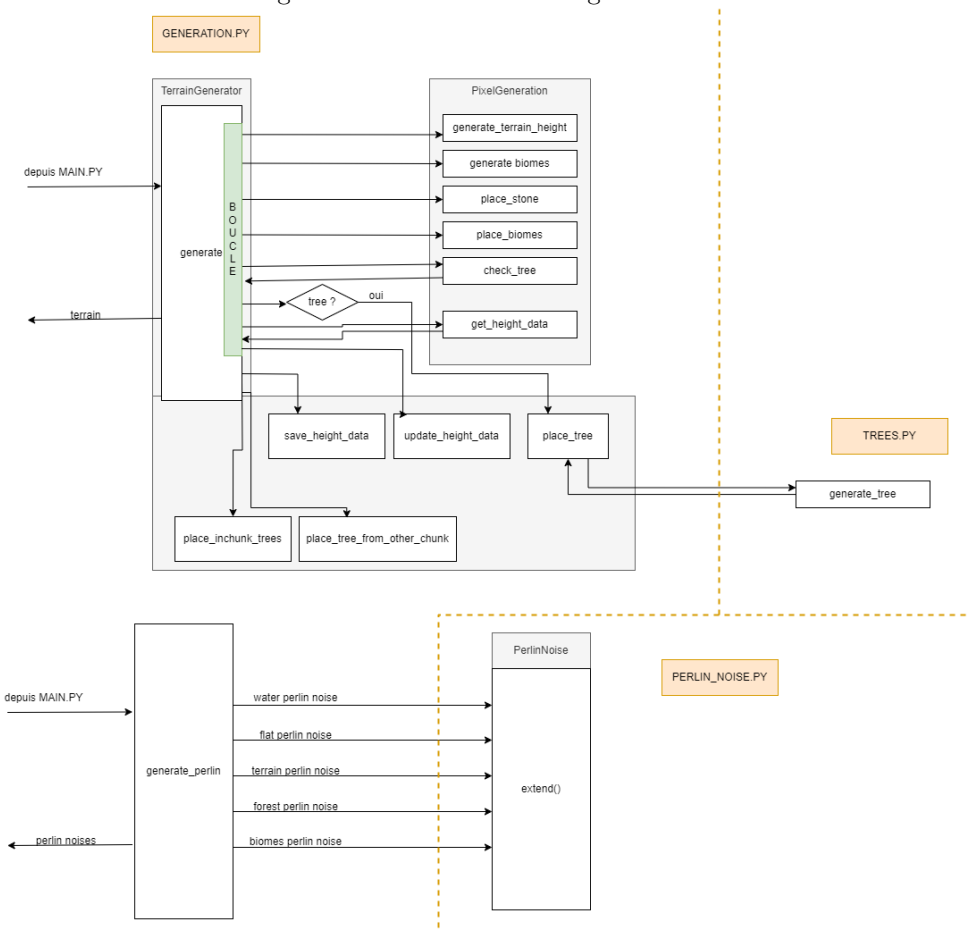
FONCTIONS DE STRUCTURES :

check_for_structures (top_left , bottom_right)
(top-left, bottom-right) : (coin haut gauche, coin bas droit), rectangle de la zone à checker pour les structures
Verifie si il y a des structures dans la zone, en faisant une boule sur les coordonnées normalisées x et y, et en verifiant avec structures.check_out_of_bounds pour chaque chunk.

3.3.2 generation.py

Fichier s'occupant de tout ce qui concerne la géneration du terrain

Figure 8: Architecture de la géneration



classe TerrainGenerator Generateur de terrain entre top-left et bottom-right, appellé par main.py.

__init__ (self , top_left , bottom_right, seed, params)
(top-left, bottom-right) : (coin haut gauche, coin bas droit), rectangle de la zone à génerer
seed : str, seed de géneration
params : liste, liste des perlin noises précédemment calculés.

generate(self)
Génère le terrain en itérant sur self .x_range et self .y_range. A chaque itération, inite la classe géneratrice d'1 pixel, qui est l'équivalent ici d'une colonne z en 3D, avec l'instance pixel_gen de PixelGeneration(). Ensuite, genere le relief (dont l'eau) si le relief est activé avec pixel_gen . generate_terrain_height (). Puis, genere les biomes (plaine, desert, neige) si le relief et les biomes sont activés avec pixel_gen .generate_biomes(). Ensuite, ajoute un bloc supplémentaire (si on est à côté de l'eau) avec pixel_gen .add_bloc().

Dans un second temps, toujours dans la boucle, place la pierre en fonction de la quantité définie plus tôt et place des minérais si les minérais sont activés avec pixel_gen.place_stone() Ensuite, place les biomes (plaine, desert, neige) si le relief et les biomes sont activés avec pixel_gen.place_biomes(). Enfin, ajoute l'eau, si le relief est activé avec pixel_gen.add_water(). Ensuite, verifie si un arbre se trouve sur ce pixel avec pixel_gen.check_tree(). Si les infos sont retournées, on place l'arbre avec self.place_tree().
Dans un troisième temps, avec la fonction pixel_gen.get_heights_info(), on obtient et met à jour toutes les infos à propos des hauteurs de la pierre, la hauteur totale, la hauteur de l'eau, ce qui évitera des boucles inutiles quand on place des volcans ou villages. Ensuite, on met à jour ces données de hauteur avec self.get_heights_info(), puis on ajoute la slice calculée à l'array principal qui contient toutes les slices, en récupérant pixel_gen.z_slice.
Enfin, après la boucle, on sauvegarde les données des hauteurs, on place les arbres qui se situent dans la chunk (ou les bouts si l'arbre est en bordure) avec self.place_inchunk_trees(). Ensuite, on place les arbres qui proviennent d'autres chunks adjacentes et qui ont un arbre en bordure avec self.place_tree_from_other_chunk(). Enfin, on update le out_of_border_tree_placeholder, afin que les arbres en bordures soient ajoutés aux chunks adjacents correspondants

place_tree(self, tree_x, tree_y, is_sand, height)
(tree-x, tree-y) : (int, int), coordonnées relatives du centre de l'arbre dans la chunk
is-sand : bool, l'arbre est il placé sur du sable
height : int, hauteur de la base de l'arbre
La fonction place l'arbre à des coordonnées relatives tree-x, tree-y, avec 'height' qui est la hauteur de la base de l'arbre. Met un arbre normal ou un palmier en fonction de s'il y a du sable.
Obtient d'abord le tree par trees.generate_tree(is_sand), puis place les blocs dans le tree-placeholder de la classe.

update_height_data(self, x, y, gen_heights_info)
(x, y) : (int, int), coordonnées relatives du pixel dans la chunk
gen-heights-info : données de la hauteur (nombre de blocs de pierre, hauteur totale, ...) Ajoute à 'self.implicated_chunks' les données de la hauteur au point (x, y)

save_height_data(self)
Sauvegarde les données de hauteur de 'self.implicated_chunks'.

Pour chaque chunk impliquée (boucle), on regarde si le fichier de heights existe déjà. Si oui, on le combine avec nos heights de self.implicated_chunks. Ensuite, on sauvegarde le fichier de heights.

place_inchunk_trees(self)
Place les arbres qui sont DANS la chunk, même s'ils dépassent (dans ce cas, les bouts d'arbres qui ne

sont pas dans la chunk sont ajoutés à une liste qui va permettre de l'ajouter à d'autres chunks (voir la fonction suivante))

place_tree_from_other_chunk(self)
Place les bouts d'arbres qui ont été générés sur la bordure d'une AUTRE chunk adjacente.

classe PixelGeneration())
Génère une slice z, à des coordonnées (x,y), controlé par le TerrainGenerator

__init__ (self , master, x, y, perlin_noises)
master : instance de TerrainGenerator à laquelle appartient cette instance, justement initiée dans l'instance master.
(x, y) : (int, int), coordonnées x, y relatives
perlin-noises : liste de dictionnaires, liste des perlin noises (=paramètres de géneration) déjà calculés qui seront utilisés pour la generation.

generate_terrain_height (self , gen_condition) gen-condition : bool, si il y a du relief
Fonction responsable de définir la hauteur du terrain à une coordonnée (x,y), et si il y a de l'eau. La gen-condition est si il y a du relief.

Simule la hauteur de l'eau et détermine si il peut y avoir de l'eau selon la formule
water_level_projection = perlin_terrain x perlin_eau $*$ 3 + WATERNESS.
Si à cette coordonnée, water_level_projection < -0.5, il y a de l'eau, alors on détermine la profondeur de l'eau. Sinon, on calcule la hauteur du terrain émergé, avec la formule
(water_level_projection + (perlin_flat x 3 − perlin_eau + perlin_biomes)
x sigm2(water_level_projection)) x WORLD_AMPLIFICATION
en accentuant la hauteur si elle est plus élevée. Si il n'y a pas de relief, fait un monde plat.

generate_biomes(self, condition) condition : bool, si il y a du relief et les biomes activés
Genere le biome à ces coordonnées, si les biomes sont activés (c'est la condition). Utilise les perlin noises suivants : biomes_perlin, flat_perlin et aussi avec la hauteur de terrain calculée précedemment.

place_stone(self , condition, condition_ores) condition : bool, si il y a le relief activé
condition-ores : bool, si la generation de minérais est activée
Place les blocs de pierre et les minérais à l'interieur, dont le nombre a été défini plus tôt. Si la generation de minérais est activée, pour chaque bloc on choisit un nombre dans des valeurs aléatoires prégénérées (pour gagner du temps, car les random.randint() sont lents), en fonction de ce nombre on peut mettre le minéai au lieu du bloc de pierre.

place_biomes(self , condition) condition : bool, si il y a du relief et les biomes activés
Place le biome (sable/herbe/neige) à ce bloc.

D'abod, definit le seuil à partir duquel la neige est génerée. On itère sur le nombre de dirt-blocs. Si ce pixel est du sable, on ajoute dans tous les cas un bloc de sable. Sinon, si c'est le premier bloc, si il y a de la neige à ce pixel, on met de la neige, sinon on met de l'herbe. Pour les autres blocs, on met de la terre. Si il n'y a pas de biomes, met seulement de l'herbe.

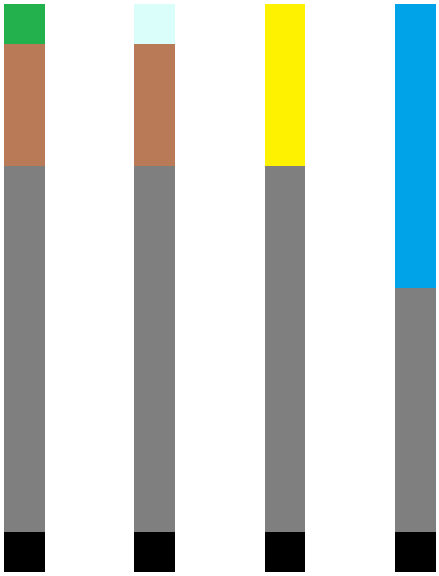
add_water(self, condition) condition : bool, si il y a du relief
Place l'eau s'il y a de l'eau (défini plus tôt) à ces coordonnées.

check_tree(self , condition) condition : bool, si il y a du relief
Retourne un bool qui indique si il y a le centre d'un arbre en ces coordonnées. La probabilité qu'il y ait un arbre est définie par le perlin_forest et par la hauteur calculée précédemment. Si on a du sable, il y aura moins de variations de densité d'arbres.

get_heights_info (self) Retourne les informations sur le nombre de blocs de pierre, la hauteur totale, et le nombre de blocs totaux excepté de l'eau.

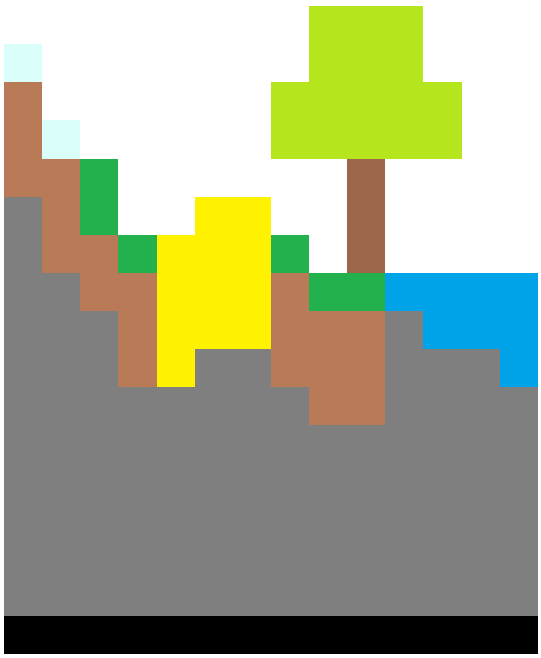
Pour plus de clarité, voici les 4 configurations de terrain possible:

Figure 9: Vue en coupe d'une colonne z type pour chaque biome



Dans l'ordre de gauche à droite, on a : plaine, neige, desert, océan. Il y a de la bedrock au fond, puis beaucoup de pierre, avec des minérais non représentés ici. Il y a ensuite, si on est emegé, de la terre, de l'herbe/neige, ou du sable. Sinon il y a de l'eau.

Figure 10: Vue en coupe d'une generation type



Nous avons ici donc de gauche à droite 2 colonnes de biome neige, puis 2 colonnes de plaine, puis 3 de desert, 3 de plaine (avec un arbre), puis 4 d'océan. L'arbre est composé de "troncs" et de "feuilles"

sigm2(x) Sigmoide responsable du fait que le terrain tend progressivement vers la hauteur de l'eau plus on est près de l'eau, selon la formule : $\frac{1}{1+2^{-20x}}$

generate_perlin(seeds, render) seeds : liste, seeds de géneration des perlin noises
render : tuple de tuple, zone sur laquelle on génère les noises
Genère tous les Perlin Noises nécessaires à la géneration de terrain.

Les paramètre du perlin noise sont : amplitude, octaves, persistence, fréquences artificielles.
- eau (océans et lacs) : 200, 0, 0.5, [0.5]
- plat/pics de relief : 150, 0, 0.5, [0.7]
- variations génerales de terrain : 120, 6, 0.6, [0.1, 0.5]
- densité des arbres : 200, 0, 0.5, [0.8]
- biomes : 200, 5, 0.5, [0.5]

On essaie de génerer le moins d'octaves / fréquences pour chaque perlin noise, à des fins d'efficacité, quitte à combiner ces différents perlin noises après.

concatenate_results(results) results : dictionnaire, contient les résultats du terrain de chacun des différents processes.
Rassemble tous les résultats des différentes zones génerées, les assemble dans un même array.

convert_into_chunks(arr, nb_chunk_x, nb_chunk_y, top_left) arr : grand array qui contient tout le terrain rassemblé
nb-chunk-x, nb-chunk-y : nombre de chunks à découper en x et en y
top-left : tuple, coin haut gauche
Convertit un array complet en chunks de taille 16x16, en le découpant.

save_chunks(nb_chunk_x, nb_chunk_y, chunks, chunks_coords, path)
nb-chunk-x, nb-chunk-y : nombre de chunks en x et en y chunks : np array, chunks découpées
chunks-coords : dictionnaire, coordonnées de bloc des chunks
path : str, chemin de sauvegarde.
Sauvegarde plusieurs chunks à un chemin donné, en itérant sur nb-chunk-x et nb-chunk-y.

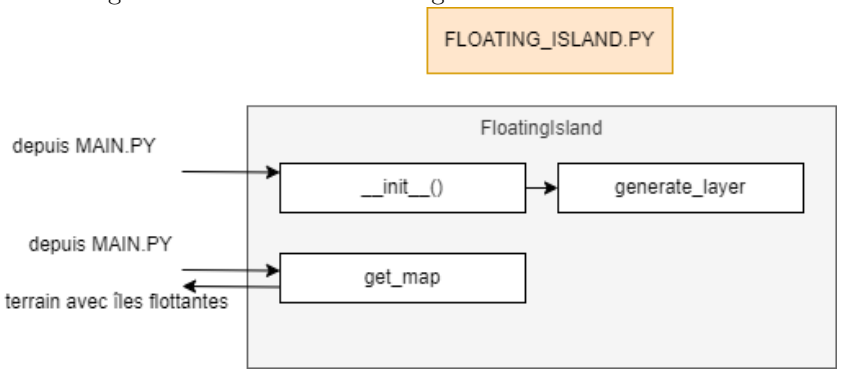
save_chunk(arr, top_left , path)
arr : np array, chunk de terrain
top-left : tuple, coordonnées de bloc de la chunks
path : str, chemin de sauvegarde.
Sauvegarde 1 chunk à un chemin donné.

3.3.3 trees.py

generate_tree(is_sand)
is-sand : bool, l'arbre est-il sur du sable ? Retourne les blocs de l'arbre, palmier si on est sur du sable, arbre normal sinon.

3.3.4 floating_island .py

Figure 11: Architecture de la géneration d'îles flottantes



classe FloatingIsland()

__init__ (self, seed, current_map, top_left, **perlin_noises)
seed : str, seed de géneration des îles flottantes
current-map: np array, terrain actuel
top-left : coin haut gauche de géneration
**perlin-noises : ensemble des perlin noises déjà génerés, qu'on utilise des perlin noises déjà génerés pour le terrain pour gagner du temps.
Classe responsable des iles flottantes présentes dans les airs. Les îles flottantes sont générées sur plusieurs couches afin d'avoir une diversité aussi en z, donc pour chaque chouche, appelle la fonction generate_layer().

self.generate_layer(self, seed, height_shift)
seed : str, seed de generation du layer/couche
height-shift : int, décalement de hauteur par rapport à la hauteur de base.
Cette fonction genère une couche d'ile flottante à une hauteur donnée.

D'abord, calcule la hauteur totale de l'île, puis calcule 3 perlin noises :
- dispersion et disposition (principal) : 80, 1, 0.5
- variations de base : 50, 0, 0.5, [4]

- petites variations (sous l'île) : 50, 0, 0.5, [8] Puis on calcule un random.randint() responsable des toutes petites variations sous l'île flottante. En itérant sur x-range et y-range, on détermine à quel point il y a une île flottante grâce au perlin-terrain, au perlin-disposition-dispersion et avec le perlin-noise-variations-de-base. Determine ensuite la hauteur de l'île flottante, puis calcule le nombre de blocs sous l'île flottante. Ces blocs sont des blocs d'herbes pour les 3 plus hauts, et de la pierre pour le reste.

3.3.5 structures.py

pregenerate(top_left , bottom_right, seeds , player_pos)
Cette fonction pregenere les positions structures lors d'un scan.

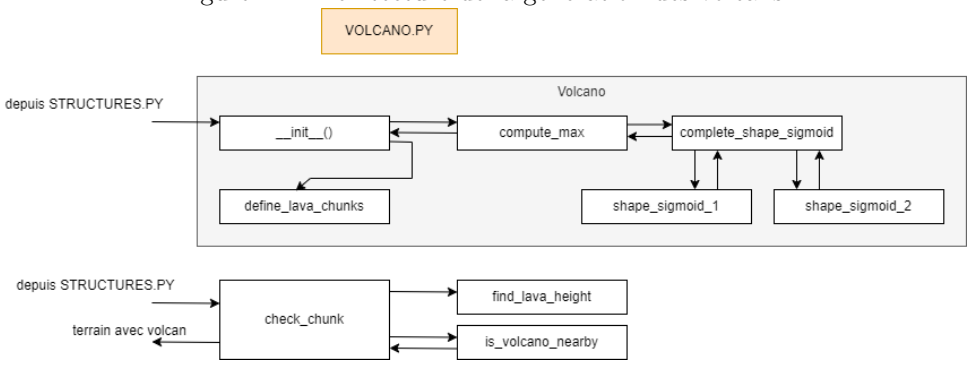
Si les volcans sont activés, genere les position des volcans en appellant generate_positions(). Si les villages sont activés, genere les position des villages en appellant generate_positions().

generate_positions(entity , top_left , bottom_right, seed , player_pos , par_1000x1000, step)
entity : classe Volcan ou Village
player-pos : tuple, position du player (x, y, z)
par-1000x1000 : int, quantité théorique de cette structure par aire de 1000x1000 blocs.
step : int, puissance de 2 de préference, espacement entre 2 checks de structures.
Cette fonction genere les positions d'une structure dans une zone qui va de top-left à bottom-right, avec une seed donnée, et un nombre theorique de structure par 1000x1000.

D'abord, calcule la chance 1/proba qu'il y ait une structure par itération, evite de faire ce calcul de manière redondante. Ensuite, test si il y a une entité à ces coordonnées. Place ensuite l'entité aux positions trouvées.

3.3.6 volcanos.py

Figure 12: Architecture de la géneration des volcans



classe Volcano()
Classe qui va génerer le volcan et sauvegarder ses données. Le volcan sera ainsi géneré seulement sur la chunk quand une chunk du volcan est checkée (check-chunk)

__init__ (self , coords, seed, player_pos)
coords : tuple, centre du volcan
seed : str, seed du volcan
player-pos : tuple, position du player
Determine la hauteur et la pente du volcan. Puis calcule la hauteur max du volcan et le rayon auquel cette hauteur est atteinte. Ensuite, calcule la distance qui sépare le centre du volcan theorique et le player. Si cette distance est assez grande, le volcan est assez loin pour être géneré sans bugs theoriques, appelle la fonction self . define_lava_chunk ()

define_lava_chunks (self))
La fonction définit les chunks sujettes à un ajout de lave, puis sauvegarde toutes les informations du volcan. Definis les chunks qui sont sur le rayon du volcan, pour ainsi savoir quelle est la hauteur minimum de ce rayon (qui dépend du terrain pas encore géneré). On va donc devoir checker la hauteur minimum pour chacun des points du rayon, pour savoir cette hauteur minimum. Quand on sait ça, on génere la lave à l'interieur jusqu'à cette hauteur, pour qu'elle ne dépasse pas du volcan.

D'abord, definit quelles chunks sont impliquées par la lave, en itérant sur theta en coordonnées polaires, donc les chunks qui sont sur le radius, et celle qui ne le sont pas mais qui ont quand meme de la lave (à l'interieur).
Determine ensuite l'ID propre du volcan.
Va ensuite sauvegarder toutes les chunks impliquées, en itérant sur les chunks impliquées. Sauvegarde toutes les données du volcan. Ensuite, update les chunks déjà générées si jamais elles existaient (ce qui n'arrive normalement pas).

compute_max(self))
Calcule le maximum de la fonction de forme du volcan.

check_chunk(current_map, top_left_)
current-map : np array, terrain actuel
Verifie si il y a de la lave à cette chunk et regarde si elle est sous l'influence d'un volcan, au quel cas il sera ajouté. Si cet endroit contient de la lave, on calcule les hauteur minimum des coordonnées à verifier pour cette chunk en appellant find_lava_height ().
On liste tous les volcans existants, et on check si la chunk est sous l'influence du volcan en appellant check_volcano_nearby().

find_lava_height (top_left_)
La fonction calcule les hauteur minimum des coordonnées à verifier (qui sont celles qui sont sur le cercle formé par le haut du volcan).

D'abord, load les hauteurs de terrain, sauvegardées dans generation.py, puis si cette chunk est bien située sur le rayon, on verifie la chunk en itèrant sur les coordonnées à check en recensant le nombre de blocs de pierre à ces coordonnées. Regarde la hauteur minimale sur ces coordonnées, puis sauvegarde le résultat de ce check.
Check si toutes les chunks ont été verifiées, si oui, determine le minimum de toutes les hauteurs de terrain au niveau du cercle du haut du volcan (définies plus haut et sauvegardées). Ensuite determine la hauteur de la lave, puis place la lave dans le volcan en appellant place_lava.

check_volcano_nearby(top_left_ , ID, current_map)
ID : str, ID du volcan à check
current-map : np array, terrain actuel
La fonction verifie pour ce volcan si il influence la chunk, si oui, on place les blocs de volcan (basalte) en conséquence.

D'abord, fais une première verification pour savoir si le volcan est assez proche, puis calcule la hauteur theorique qu'aurait le volcan à cette chunk si on prenait le centre de la chunk. Si cette hauteur est suffisante, on itère sur x et y, on calcule la hauteur du volcan à ce point. Pour chaque bloc (x,y), si la hauteur est suffisante, on remplit la colonne z de basalte (aux coordonnées current_map[x, y, :] , depuis la pierre jusqu'à la hauteur du volcan. Si ces coordonnées sont assez proches pour qu'il y ait de la lave, sauvegarde ensuite le fait qu'il y ait de la lave à ces coordonnées.
Sauvegarde ensuite.

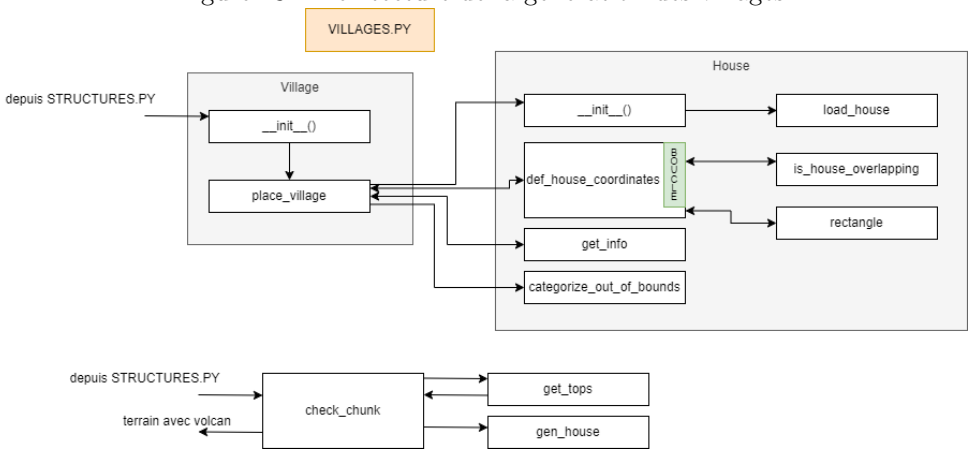
place_lava (ID, lava_height , radius_chunks, inner_chunks)
ID : str, ID du volcan sur lequel on place la lave.

lava-height : int, hauteur de la lave à placer
radius-chunks : dict, chunks situées sur le radius du volcan (avec de la lave)
inner-chunks : liste, chunks situées à l'interieur du radius (avec de la lave)

D'abord, ajoute les chunks interieures (inner-chunks) aux chunks qui ont de la lave, puis ajoute les chunks du rayon (radius-chunks), puis supprime les doublons.
Itère sur les chunks qui ont de la lave, load la chunk et les heights. Itère en x et y, et si il y a de la lave à cet endroit (sauvegardé plus tôt), rempls la colonne de lave, depuis le basalte jusqu'à la lava-height. Sauvegarde ensuite la chunk modifiée. Ensuite open la data du volcan à cette chunk, et supprime le volcan de la liste des volcans. Si il reste des volcans, save le file avec le volcan supprimé.
Si il ne reste plus de volcan dans la chunk, supprime le file de chunk de volcan, et ajoute les chunks à l'array-des-coordonnées-des-chunks-terminées. Ensuite, save cet array-des-coordonnées-des-chunks-terminées, que le data_manager_generation.py va charger, il va ensuite envoyer cette information (que la chunk est terminée).

3.3.7 villages .py

Figure 13: Architecture de la géneration des villages



classe Village()
Classe qui va génerer le village et sauvegarder ses données
Le village sera ainsi généré seulement sur la chunk quand une chunk du volcan est checkée (check-chunk)
Cependant, le village ne sauvegarde que les maisons individuelles, et pas le village en tant qu'entité.

__init__(self, coords, seed, player_pos)
coords : tuple, coordonnées du village
seed : str, seed du village
player-pos : tuple, position du player
Calcule la distance qui sépare le centre du village du joueur. Si le village est suffisament loin du player, on peut génerer. On place le village en appellant self.place_village().

place_village(self)
La fonction place le village et ses maisons.

D'abord, determine d'abord le nombre de maisons que le village aura. Puis ouvre la liste des types de maison, et calcule les probabilités pour chacune des maisons de spawner.
Genere d'abord la tour centrale du village, puis ajoute toutes les autres maisons, en itérant sur le nombre de maisons.
Dans la boucle, choisis le type et l'orientation de la maison, crée l'instance de maison avec House(). Definis ensuite les coordonnées de la maison dans le village avec house.def_house_coordinates(). Obtiens le rectangle dans lequel se situe la maison avec house.get_info(). Sauvegarde enfin la maison pour être utilisée ensuite par check-chunk, avec house.categorize_out_of_bounds()

classe House()
Classe responsable du placement d'une maison et de sa sauvegarde, dépendant directement de son village parent.

__init__ (self , house_id, master_village : Village)
house-id : str, id de la maison.
master-village : instance de village auquel la maison appartient.
Load la maison selon le nom et l'orientation donnés, puis definit le rectangle relatif de la maison.

def_house_coordinates(self)
Cette fonction definit la position et le placement de la maison en fonction des autres maisons du village.

Simule des positions de la maison jusqu'à ce qu'une position convienne, en itérant sur les coordonnées polaires. Dans la boucle, simule une position, regarde si la maison est assez loin des autres. Si oui, definis la position de la maison à cet endroit.

categorize_out_of_bounds(self)
Cette fonction verifie à quel chunks appartiennent les blocs de la maison, puis sauvegarde ces infos. Le but est de trouver quelle est la hauteur maximum de terrain pour la maison, afin de remplir de pierre en dessous. Il faut donc que TOUTES les chunks auquelles appartient la maison soient génerées avant que la maison puisse être placée.

D'abord, definis les chunks dans lesquels se trouve la maison (1, 2 ou 4). Ensuite, ajoute les blocs de la maison dans les chunks correspondantes. Definis après les chunks pour lesquels on doit checker la hauteur maximale du terrain. Donne ensuite un id à la maison.
Sauvegarde ensuite la data de la maison, puis ajoute les bouts de la maison aux fichiers de chunks en itérant sur les chunks impliquées.
Enfin, update les chunks déjà génerées si jamais elles existaient (ce qui n'arrive normalement pas).

rectangle(self)
Retourne le rectangle, ou plutôt le parallelepipede (en 3D), dans lequel se situent les blocs de la maison.

is_house_overlapping(self , house_a_absolute_position , houseB, min_gap)
house-a-absolute-position : cordoonées absolues du centre de la maison actuelle.
houseB : tuple, (houseB top-left, houseB bottom-right)
min-gap : espacement minimum entre deux maisons

Cette fonction retourne le rectangle dans lequel se situent les blocs de la maison.

D'abord, Determine les coordonnées absolues de la maison, la taille theorique de la maison est augmentée d'un min-gap, pour éviter que les maisons soit collées. Determine ensuite si les maisons s'intersectent, ou sont trop proches, en appellant intersect () pour check si les deux rectangles de maisons d'intersectent.

load_house(self)
Cette fonction load la maison, depuis le template stocké en local, vers la liste blocs-list.

D'abord load le fichier numpy, puis convertis l'array 3D en liste de valeurs non nulles associées à un (x, y, z).

check_chunk(current_map, top_left_)
current-map : np array, terrain actuel
Cette fonction erifie si il y a des maisons à cette chunk. Si une maison est terminée, on peut la placer.

D'abord, i il y a au moins une maison sur cette chunk, load les données de la chunk. Ensuite itère sur le nombre de maisons qui se situent dans la chunk. Dans la boucle, load les données de la maison, verifie que la chunk fait bien partie de la maison. Puis si la maison n'a pas déjà été verifiée pour cette chunk, regarde la hauteur du terrain sous la maison en appelant get_tops(). Cette chunk est à présent verifiée pour cette maison.
Ensuite, verifie si toutes les chunks de la maison ont été verifiées. Si c'est le cas, genère la maison en appelant gen_house().
Si la maison n'a pas été génerée, on sauvegarde les informations de la maison.

gen_house(tops, ID)
tops : dictionnaire qui contient toutes les hauteurs maximales de terrain pour les différentes chunks de la maison.
ID : str, id de la maison
Une fois que l'on sait la hauteur maximale du terrain sous la maison dans toutes les chunks de la maison, cette fonction peut generer la maison sur ces chunks.

D'abord, trouve le point le plus haut sous la maison (le max des tops), qui sera la hauteur de la maison. Supprime le fichier de la maison, car elle est placée. Itère ensuite sur les tops (=chunks où est la maison). Dans la boucle, load les données de la chunk (à laquelle la maison appartient), puis recupère les blocs de la maison situés dans ce chunk. Marque la maison comme finie dans le chunk, on la supprime. Si il reste encore des maisons dans la chunk, la chunk n' donc est pas encore terminée, on sauvegarde le fichier de chunk Sinon, on supprime le file de chunk de village, et ajoute les chunks à l'array-des-coordonnées-des-chunks-terminées. Ensuite, save cet array-des-coordonnées-des-chunks-terminées, que le data_manager_generation.py va charger, il va ensuite envoyer cette information (que la chunk est terminée).

Ensuite, toujours dans la boucle, load le terrain à cette chunk. Pour chaque bloc de maison, remplis la colonne z de pierre sous la maison, puis ajoute le bloc de maison à la chunk. Sauvegarde enfin la chunk.

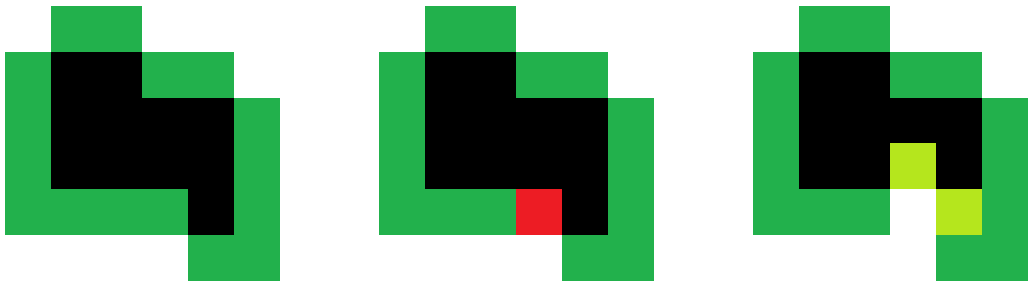
get_tops(mins_maxs, current_map, top_left_)
mins-maxs : tuple de tuple, du format (minX, minY), (maxX, maxY), qui correspond au rectangle formé par le bout de maison dans la chunk.
current-map : terrain actuel
Definis le point le plus haut du terrain dans une zone donnée.

D'abord, créee deux arrays d'étendue (x et y) (équivalent de "range()"). Ensuite, on prend la map actuelle, on la slice sur les zones des deux arrays d'étendue, puis on fait un masque pour les blocs ignorés (= un array qui va contenir un 1 là où les blocs sont ignorés (ex: feuilles), et 0 là où ils ne le sont pas (ex: pierre)). Ensuite, on trouve l'indice z du premier bloc qui est ignoré pour chaque coordonnée, qui sera la hauteur à ces coordonnées.
On définit ensuite la hauteur maximale de toutes les hauteurs checkées (dans la slice), qu'on retourne.

3.3.8 fast_renderables .py

Notre but ici va être de calculer les renderables, mais pour la modification de seulement un seul bloc (le bloc cassé). Ce serait donc inutile de recalculer les renderables de toute la chunk (comme ce que l'on fait dans renderables.py).
Pour chaque bloc cassé, on va donc ajouter aux renderables (s'ils n'existent pas déjà) tous les blocs autour du bloc cassé, car ils sont à présent visibles.

Figure 14: Exemple 2D de mise à jour des renderables pour un bloc cassé



On a au début les blocs verts qui sont ceux renderables, et les blocs noirs non renderables (1). Imaginons qu'on casse ensuite le bloc rouge (2). Deux blocs seront alors ajoutés aux renderables , un bloc sera supprimé (celui qui a été cassé) (3)

cassage(coords, chunk_coords
Determine les changes dans les renderables qu'impactent ce cassage de bloc. Appelle la fonction add_renderables(*coords, *chunk_coords). Retourne un ensemble (set())des changements retournées, afin d'éviter les duplicats.

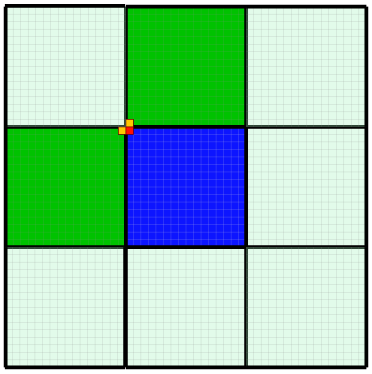
add_renderables
Determine les nouveaux renderables après une suppression de bloc en x, y, z.
D'abord, supprime des renderables le bloc cassé, puis verifie les blocs au dessus et en dessous sur l'axe Z.
On a trois cas possibles pour chacun des axes x ou y. Pour l'axe x:
- Si x== 0, on est tout à gauche de la chunk, on update le bloc le plus à droite de la chunk adjacente à gauche (donc à gauche du bloc cassé) et le bloc à droite du bloc cassé
- Si x == settings.CHUNK_SIZE - 1, on est tout à droite de la chunk, on pdate le bloc le plus à gauche de la chunk adjacente à droite (donc à droite du bloc cassé) et le bloc à gauche du bloc cassé
- Sinon, on est n'est pas sur le bord d'une chunk pour l'axe x, on update tout simplement les blocs à droite et à gauche du bloc cassé.

Pour l'axe y, c'est très similaire (ici, dessus et dessous font réference à l'axe y et non l'axe z):
- Si y == 0, on est tout en haut de la chunk (sur l'axe y), on update le bloc le plus en bas de la chunk adjacente en haut (donc au dessus du bloc cassé) et le bloc au dessous du bloc cassé.
- y == settings.CHUNK_SIZE - 1, on est tout en bas de la chunk (sur l'axe y), on update le bloc le plus en haut de la chunk adjacente en bas (donc au dessous du bloc cassé) et le blocs au dessus du bloc cassé.

- Sinon, on est n'est pas sur le bord d'une chunk pour l'axe y, on update tout simplement les blocs au dessus et au dessous du bloc cassé.

Ici, quand je dis "update", cela veut dire qu'on appelle la fonction update_renderable_bloc()

Figure 15: Plan vue de haut de 3x3 chunks



Par exemple, ici, on est dans le cas où x et y sont égal à 0. On doit donc mettre à jour les blocs tout autour du bloc cassé (en rouge), ce qui comprend deux blocs d'autres chunks (en orange). On devra donc utiliser ces chunks (en vert foncé).

update_renderable_bloc(shift , position , chunk_x, chunk_y, changes, delete=False)
shift : tuple, décalage (x, y) de chunks par rapport à la chunk initiale (un shift de (-1, 0) correspondra à la chunk à gauche de la chunk initiale).
position : position relative du bloc dans la chunk (x, y, z) chunk_x, chunk_y : coordonnées de la chunk initiale changes : liste à laquelle va être ajouté le changement (suppression ou ajout d'un bloc renderable), si il y en a un. delete : bool, si le bloc sera supprimé des renderables La fonction ajoute ou supprime le bloc des renderables s'il ne l'est pas déjà. D'abord, obtient pour la chunk les blocs renderables et les blocs de chunk. Definit ensuite le type de bloc à être ajouté, puis check si il y a déjà un bloc renderable à ces coordonnées dans la liste.
Si il n'y en a pas déjà, ajoute le bloc à la fin des renderables
Sinon, supprime les renderables déjà existants à ces coordonnées. Si le bloc doit être supprimé, on s'arrête là. Sinon, si il doit être ajouté, on ajoute le bloc à la fin des renderables.
On verifie ensuite si il y a eu un changement, c'est à dire si les renderables sont restés inchangés ou pas. Si il y a eu un changement, on appelle data_manager.change_in_renderables() pour indiquer que il y a eu un changement dans les renderables, qui s'occupera de prendre en compte et sauvegarder ces changements. Ensuite, on ajoute à 'changes' ce que la fonction handle_new_renderable() renvoie (cette fonction va formater ce qu'on doit mettre dans changes selon le format attendu par Oscar).

handle_new_renderable(is_added, chunk_x, chunk_y, bloc_pos, bloc_value=0)
Les paramètres sont les mêmes que si dessus, is_added = not delete, et bloc_value est le type de bloc à ajouter/supprimer. Fonction appelée pour savoir quoi faire avec un nouveau bloc renderable, et le format

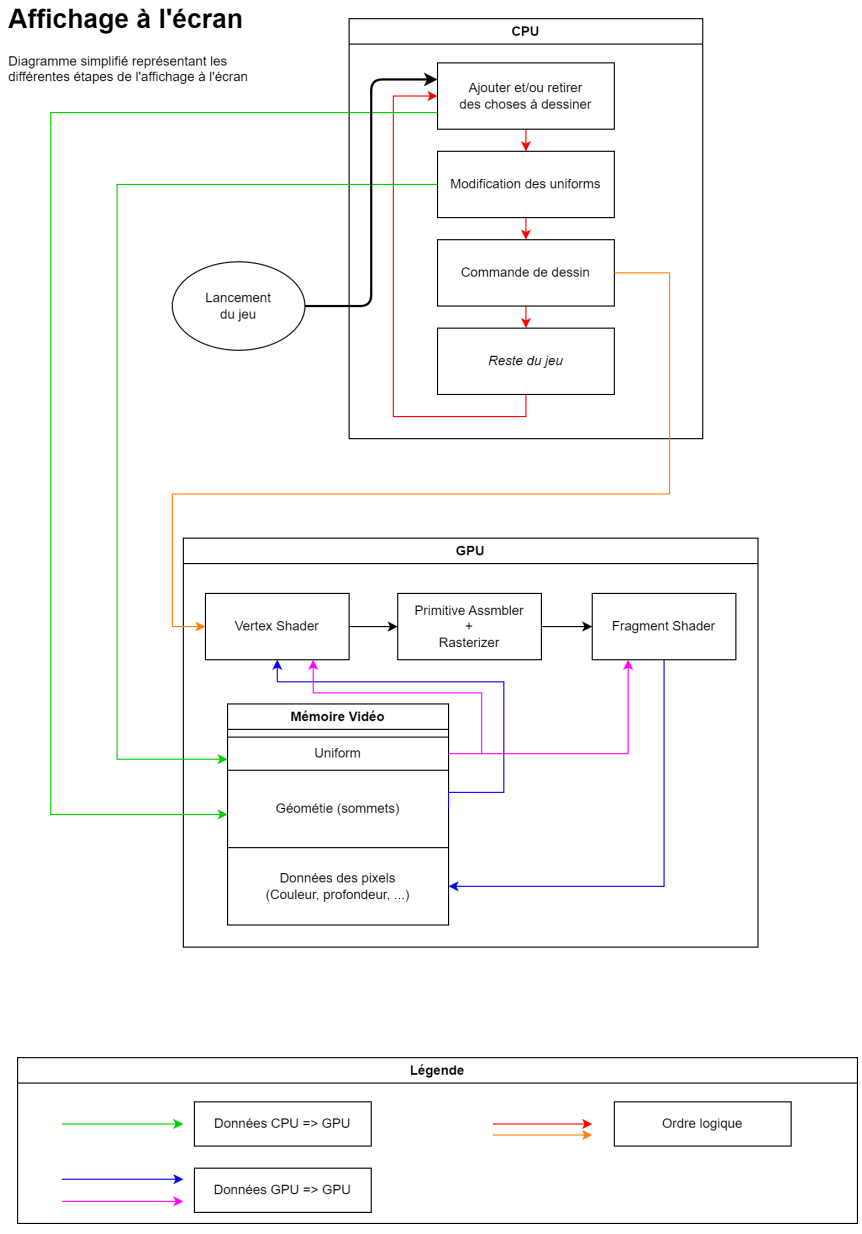
de ce qui sera ajouté à changes. Le format de ce qui sera ajouté à 'changes' est donc le suivant : (chunk_x, chunk_y), bloc_pos, bloc_value

3.3.9 settings .py

4 Graphismes

4.1 OpenGL

Nous utilisons l'API OpenGL pour pouvoir afficher les graphismes à l'écran à de haute performances. Dans cette section le vocabulaire spécifique est présenté ainsi que le fonctionnement général de OpenGL.



Shader : Un programme éxécuté sur le GPU. Les termes 'shader' et 'programme GPU' seront utilisé de manière interchangeable dans cette partie.

Vertex shader : C'est une sous-partie d'un shader. Il s'agit du programme qui calcule la position à l'écran des différents éléments de géométrie (points, lignes, triangles).

Fragment shader : Parfois appelé pixel shader, c'est la sous-partie du shader qui calcule la couleur de chaque pixel.

Uniform : C'est une variable utilisé par un shader qui est identique pour chaque invocation du shader. Elle ne peut être changée que par le CPU avant la commande de dessin.

Invocation du shader : à chaque fois que le shader est éxécuté, on dit également qu'il est invoqué.

Vertex Buffer Object (vbo) : Il s'agit d'un morceau de mémoire GPU qui contient les données liées à la géométrie des objets que l'on souhaite afficher.

Index Buffer Object (ibo) : Parfois appelé 'Element Buffer Object' (ebo), il s'agit d'un morceau de mémoire GPU qui indique à OpenGL dans quel ordre dessiner les sommets. Il est facultatif. Dans le cas où il n'est pas utilisé, OpenGL dessine les éléments dans l'ordre dans lequel les sommets sont dans le vbo.

Vertex Array Object (vao) : Il s'agit d'une structure OpenGL qui lie le vbo, l'ibo et le programme GPU.

face culling : technique qui permet de ne pas faire de calculs sur faces de géométries qui ne sont pas orientés vers le joueur en les ignorant.

4.2 Fonctionnement

4.2.1 Bibliothèques et Initialisation

Bibliothèques : Pygame et ModernGL

Pygame

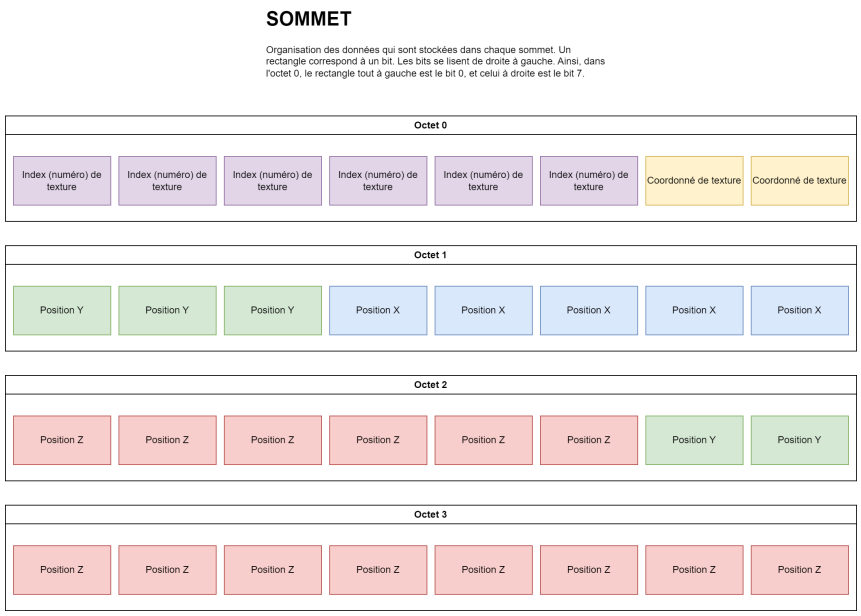
Pygame est une bibliothèque python très populaire basée sur SDL. Nous utilisons la bibliothèque pygame pour créer un contexte graphique, une fênetre mais aussi pour récupérer les entrées claviers de l'utilisateur.

ModernGL

ModernGL est bibliothèque graphique qui permet une utilisation facile des fonctionnalités OpenGL et des performances de la carte graphique. C'est pour cela que nous l'utilisons.

4.2.2 Calculs relatifs aux sommets des cubes

Les données que contient chaque sommet sont calculés avec de la manière suivante :
Les sommets sont calculés 1 à 1.
1 : Calcul de la position du sommet. On ajoute aux coordonnées du coin inférieur gauche arrière du cube un décalage précalculé qui se situe dans une table nommée 'VERTEX_XYZ_COORD_LOOKUP'.
2 : On obtient la coordonnée de texture qui correspond à ce sommet à l'aide d'une table précalculée nommée 'VERTEX_TEX_COORD_LOOKUP'.
3 : A l'aide de décalages de bits "vers la gauche", on compacte dans 4 octets (32 bits, numéroté de 0 à 31) les positions, les coordonnées de texture, et le numéro de texture. Le format est le suivant :
La coordonnée 'z' du cube est stockée du bit 18 au bit 31 inclus.
La coordonnée 'y' du cube est stockée du bit 13 au bit 17 inclus.
La coordonnée 'x' du cube est stockée du bit 8 au bit 12 inclus.
Le numéro de texture du cube est stockée du bit 2 au bit 7 inclus.
Les coordonnées de texture du cube sont stockées dans les bits 0 et 1.



4.2.3 Affichage à l'écran

Interface Graphique

Les éléments de l'interface sont dessinés en premier.
Ils sont :
-La croix centrale qui permet de savoir quel est le bloc regardé. Sa position est toujours la même mais sa taille s'adapte à celle de l'écran en fonction du uniform 'AspectRatio'.
-La 'hotbar', c'est la barre en bas de l'écran qui permet de séléctionner un bloc à placer. C'est un rectangle auquel on applique l'atlas de texture.
-Le carré de sélection, c'est le carré qui permet de savoir quel type de bloc est actuellement sélectionné. Il est dessiné avec quatre lignes qui sont décalées en fonction du uniform 'ScreenOffset'
Les différents éléments sont placés "à l'avant" du buffer de profondeur (z-buffer ou depth buffer), ce qui fait qu'ils sont toujours visibles.

Affichage des blocs

Les blocs sont dessinés dans le mode 'GL_TRIANGLE_STRIP', qui forme un triangle à partir des deux derniers sommets déjà utilisés et du prochain sommet du vbo. L'idée est de diminuer l'utilisation de mémoire en réutilisant les mêmes sommets. Les sommets sont données dans un ordre spécifique pour que le face culling fonctionne et que les textures s'affichent correctement.

Dans le vertex shader, plusieurs étapes ont lieues.
1 : Décompresser les différentes données avec des décalages de bits "vers la droite".
2 : La position du sommet et reconstitué sous forme de vecteur. Le sommet contient une position relative à celle du tronçon du cube. Les tronçons ont pour dimensions 16*16, et leur position en coordonnées de tronçons est contenue dans le uniform 'chunk_coordinates'dans donc la position absolue du sommet est :
x : sommet.x + chunk_coordinates.x * 16
y : sommet.y + chunk_coordinates.y * 16
La coordonnée 'z' est absolue.
3 : On calcule la position dans l'espace de l'écran à l'aide de matrices qui sont détaillés dans la section 5.4.
4 : On obtient à partir de tables précalculées trois nombres qui vont permettre de calculer la normale de chaque face sans utiliser de branches (if), puisque les branches peuvent êtres très lentes sur le GPU.
5 : La position, les nombres obtenus à l'étape 4, les coordonnées de textures et le type de texture sont transmis à la suite du programme.

Dans le fragment shader :
1 A l'aide du numéro de texture et des coordonnées de texture, on obtient la couleur brute du pixel.
2 On calcule le produit scalaire entre la normale de la face et la source de lumière. Cela permet de calculer la lumière qui s'applique au pixel (note : on ajoute toujours un peu de lumière d'éclairage ambiant pour éviter qu'un bloc ne soit complètement noir). On multiplie la couleur du bloc avec sa luminosité, comprise entre 0 et 1.

4.3 Implementation

4.3.1 renderer.py

Classe : Renderer : une classe qui gère les graphismes.

méthode : __init__(self)
Initialise pygame si ce n'est pas déjà fait.
Indique que le contexte graphique OpenGL doit être de version '3.3'.
Active les fonctionnalités coeur OpenGL.
Associe le contexte OpenGL et le contexte moderngl.
Active les fonctionnalités spécifiques.

méthode : bind_scene(self, scene)
scene : la scène qui doit être liée avec cet objet 'Renderer'.
Fonction pour lier une scène à cet objet 'Renderer' qui va être affiché à l'écran.

méthode : render(self)
Dessine la nouvelle image à l'écran. Applique d'abord la couleur de fond, puis affiche le monde en appelant la méthode 'render' de la scène actuellement liée.

méthode : end(self)
Cette fonction est appellée lorsque le jeu se termine. Elle libère les resources et ferme le programme.

4.3.2 shader_manager.py

Classe : ShaderManager : une classe qui gère les programmes GPU (shaders).

méthode : __init__(self, ctx)
ctx : Le contexte graphique OpenGL.

méthode : __getitem__(self, shader_name)
shader_name : Le nom du programme GPU auquel on veut accéder.
Permet d'utiliser la syntaxe '[ ]' pour accéder aux shaders contenus dans le ShaderManager.
Exemple : pour accéder à un programme nommé "test", on peut écrire :
programme = ShaderManager[test]
au lieu de :
programme = ShaderManager._programs[test]

méthode : create_program(self, program_name, vertex_shader, fragment_shader)
Fonction pour créer un programme GPU nommé 'program_name'.
vertex_shader : le code du vertex shader OpenGL.
fragment_shader : le code du fragment shader OpenGL.

méthode : delete_program(self, program_name)
Supprime le programme nommé 'program_name'.

méthode : delete_all(self)
Supprime tous les programmes GPU.

méthode : update_uniform(self, shader_name: str, uniform: str, data)
Fonction pour modifier un seul uniform.
shader_name : Le nom du programme qui possède l'uniform 'uniform' à modifier.
data : La (les) nouvelle(s) donnée(s) de l'uniform.

méthode : update_uniforms(self, shader_name: str, uniforms: list[str], data)
Modifier plusieurs uniform d'un même programme GPU.
shader_name : Le nom du programme qui possède les uniforms 'uniforms' à modifier.
data : une liste (list) ou un tuple qui contient les différentes nouvelles données à placer dans les uniforms.

4.3.3 scene.py

Classe : Scene : Une classe qui contient l'ensemble des objets qui vont être dessinés

méthode : __init__(self)
Definit les attributs de l'objet.

méthode : add_object(self, obj)
Fonction pour ajouter un objet à l'ensemble des objets sans fonctionnalités particulières qui vont êtres dessiné à l'écran. Elle renvoie un index qui permet d'identifier l'objet ajouté.

méthode : add_chunk_batch(self, chunk_batch)
Fonction pour ajouter un objet 'ChunkBatch' à l'ensemble des objets 'ChunkBatch' qui vont êtres dessiné à l'écran. Elle renvoie un index qui permet d'identifier l'objet ajouté.

méthode : add_water_batch(self, water_batch)
Fonction pour ajouter un objet 'WaterBatch' à l'ensemble des objets 'WaterBatch' qui vont êtres dessiné à l'écran. Elle renvoie un index qui permet d'identifier l'objet ajouté.

méthode : delete_object(self, index)
index: l'index qui correspond à l'objet à supprimer. Il n'apparaitra plus à l'écran.

méthode : delete_chunk_batch(self, index: int)
index: l'index qui correspond au 'ChunkBatch' à supprimer. Il n'apparaitra plus à l'écran.

méthode : delete_water_batch(self, index: int)
index: l'index qui correspond au 'WaterBatch' à supprimer. Il n'apparaitra plus à l'écran.

méthode : sort_waters(self, player_chunk)
Fonction pour ranger les 'WaterBatch' par distance décroissante avec le joueur.
Les 'WaterBatch' doivent être dessinés du plus loin au plus proche pour que la transparence fonctionne correctement.

fonction : calculate_ditance_squared(water_batch)
Fonction locale à la méthode sort_waters(self, player_chunk), qui calcule la distance au carré entre un 'WaterBatch' et le joueur.

méthode : render(self)
Dessine tous les objets sur l'écran. Dessine d'abord les objets génériques, puis les 'ChunkBatch's et enfin les 'WaterBatch's

méthode : release(self)
Supprime tous les objets de la scène et réinitialise ses attributs.

4.3.4 batch.py

Classe : Batch : Une classe qui lie des données et des programmes GPU.

méthode : __init__(self, ctx, shader, buffer_size, vertex_format, attributes, index_element_size, vertices_per_object, dynamic=True)
ctx : Le contexte graphique OpenGL.
shader : Un programme GPU.
buffer_size : Le nombre d'octets à réserver dans le vbo.
vertex_format : Le format de données que va recevoir le GPU.
attributes : Les attributs présents dans le programme GPU.
index_element_size : La taille en octets des nombres présents dans l'ibo.
vertices_per_object : Le nombre de sommets par élément de géométrie.
dynamic : Considérer ou non la mémoire GPU comme dynamique.

méthode : create_ibo(self)
Créer l'ibo dans la mémoire GPU. L'ibo indique quels sommets doivent être dessinés et dans quel ordre. Il est facultatif.

méthode : create_vbo(self)
Créer le vbo dans la mémoire GPU. Le vbo contient les données des sommets.

méthode : create_vao(self)
Créer le vao dans la mémoire GPU. Le vao est est l'objet qui lie le vbo, l'ibo, et le shader.

méthode : update_vbo(self, data, offset: int)
Fonction pour modifier les données contenues dans le vbo, écrit 'data' dans le GPU, à l'endroit indiqué par 'offset'.

méthode : update_ibo(self, data, offset: int)
Fonction pour modifier les données contenues dans l'ibo, écrit 'data' dans le GPU, à l'endroit indiqué par 'offset'.

méthode : release(self)
Libère les resources utilisées par l'ibo, le vbo et le vao. Cet objet 'batch' est supprimé.

méthode : render(self)
Afficher ce 'batch' à l'écran.

4.3.5 chunk_batch.py

Classe : ChunkBatch : Une classe qui permet d'afficher des blocs non transparents.

méthode : __init__(self, ctx, program, chunk_coord, number_of_cubes)

ctx : Le contexte graphique OpenGL.
program : Un programme GPU.
chunk_coord : Les coordonées du tronçon que ce 'ChunkBatch' représente.
number_of_cubes : Le nombre maximum de cubes que ce 'ChunkBatch' peut contenir.

méthode : add_cube(self, cube_x, cube_y, cube_z, tex)

cube_x, cube_y, cube_z : représentent les coordonnées du coin inférieur gauche arrière du cube. Cette position est relative au tronçon.
tex : le numéro de texture = le type du bloc.
Si de l'espace dans la mémoire est disponible "entre" deux cubes déjà existants, le cube sera placé à cet endroit pour éviter de fragmenter la mémoire et de la de gaspiller. Renvoie un index spécifique à ce bloc.
Le fonctionnement détaillé du calcul des données que contient chaque sommet se situe dans la section 4.2.2.

méthode : push_cubes(self, cubes_data)

Si de l'espace dans la mémoire est disponible "entre" deux cubes déjà existants, les cubes ne seront pas placé à cet endroit pour des raisons de performances.
cubes_data : un tuple ou une liste de cubes sous la forme (cube_x, cube_y, cube_z, tex).
Les coordonées (cube_x, cube_y, cube_z) représentent le coin inférieur gauche arrière du cube.
tex : le numéro de texture ¡=¿ le type du bloc.
Renvoie une liste d'indices qui représentent les blocs ajoutés.
Le fonctionnement détaillé du calcul des données que contient chaque sommet se situe dans la section 4.2.2.

méthode : edit_cube(self, cube_x, cube_y, cube_z, tex, index)

Fonction pour modifier les données d'un cube spécifié par son index. Les coordonées (cube_x, cube_y, cube_z) représentent le coin inférieur gauche arrière du cube.
tex : la coordonné de texture dans l'atlas de textures.
Procéder ainsi est plus efficace que de supprimer puis recréer un cube.
Le fonctionnement de cette fonction est le même que celui de la fonction 'add_cubes', sauf qu'au lieu de retourner l'index d'un nouveau cube, rien n'est retourné, et les données sont écrites à l'emplacement spécifié par 'index'.

méthode : delete_cube(self, index)

Supprime le cube à l'index spécifié, en écrivant des zéros dans la mémoire GPU.
Cela à pour effet de fragmenter la mémoire GPU, autrement dit, un morceau de mémoire localement continu sera séparé en deux, et de l'espace sera disponible entre ces deux blocs de mémoire.
La function 'add_cube' lutte contre cet effet.

méthode : render(self)

Affiche cet objet 'ChunkBatch' à l'écran.
Dans un premier temps, l'uniform 'chunk_coordinates' est mis à jour pour que cet objet 'ChunkBatch' soit bien placé.
Puis envoie l'instruction de dessin au GPU.

4.3.6 water_batch.py

Classe : WaterBatch : Une classe qui permet d'afficher des blocs transparents.

La classe 'WaterBatch' est extrêment similaire à la classe 'ChunkBatch'. C'est pourquoi cette classe est un copié-collé de 'ChunkBatch', avec quelques modifications. Garder deux classes différentes a l'avantage de permettre plus de changements et de modulabilité.
Les deux classes partagent les mêmes fonctions. Pour plus de détails voir la classe 'ChunkBatch'.

4.3.7 gui.py

Classe : Gui : Une classe qui affiche les éléments de l'interface utilisateur

méthode : __init__(self, ctx: line_program, hot_bar_program, selection_square_program)
ctx : Le contexte graphique OpenGL.
line_program, hot_bar_program, selection_square_program : Les différents programmes GPU.
Initialise les 'vbo's 'vao's pour chaque éléments de l'intérface.

méthode : draw_cross(self)
Dessine la croix centrale.

méthode : draw_hot_bar(self)
Dessine la hotbar.

méthode : draw_selection_square(self)
Dessine le carré de séléction.

méthode : render(self)
Dessine tous les éléments de l'interface utilisateur.

méthode : release(self)
Libère les ressources alloués pour les éléments de l'interface utilisateur.

4.3.8 texture.py

Classe : TextureAtlas : Une classe qui permet de charger et d'utiliser une texture.

méthode : __init__(self, ctx: Context, filepath)
ctx : Le contexte graphique OpenGL.
filepath : Le chemin du fichier de la texture.

méthode : use_texture(self, location)
Fonction qui permet d'indiquer quelle texture utiliser lors de l'affichage.
location : La localisation de la texture dans le contexte OpenGL.

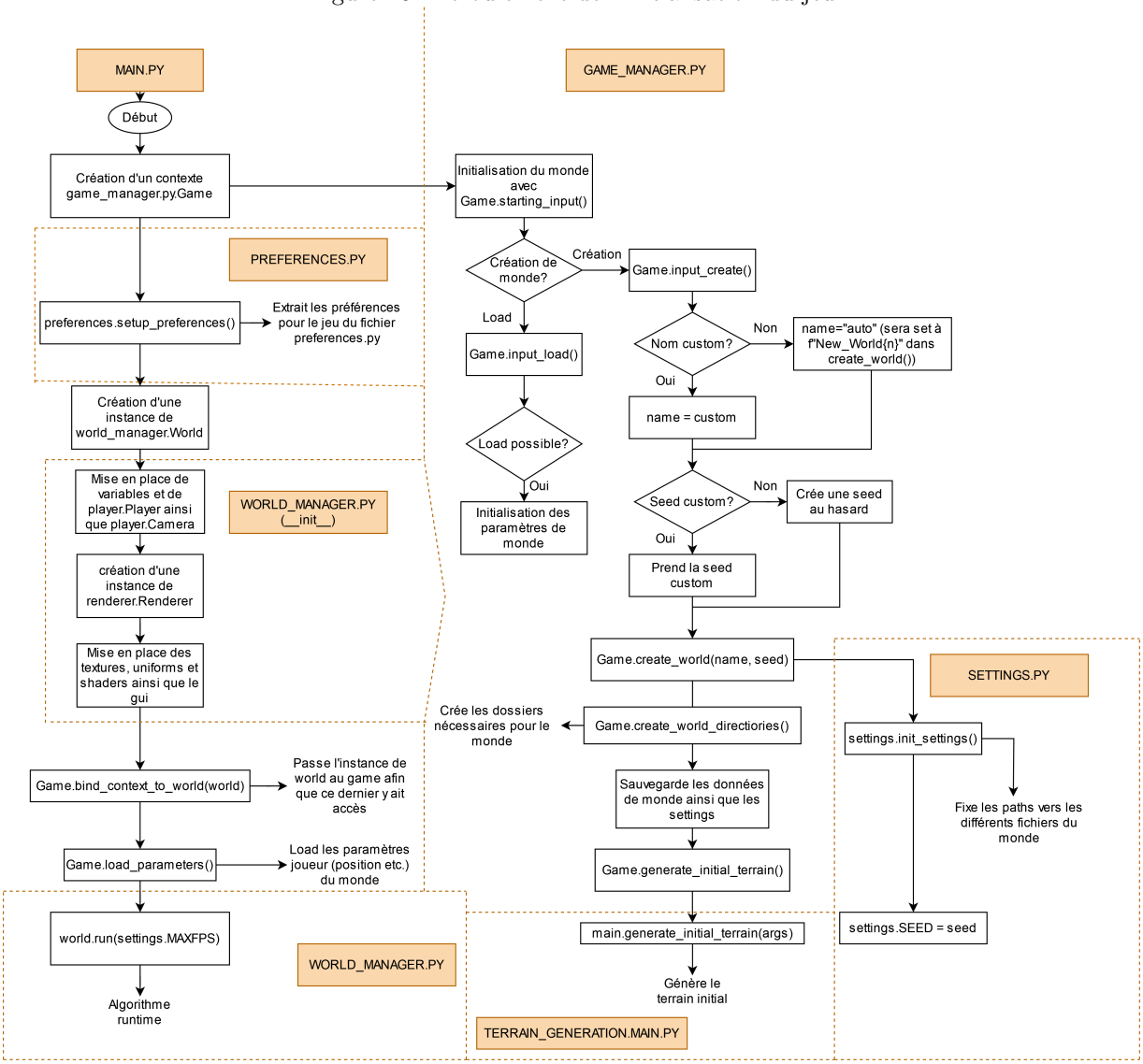
5 Le jeu

5.1 Introduction

Nous avons jusqu'à présent vu comment le terrain du jeu était généré et comment on pouvait afficher celui-ci. Ici nous discutons de la façon de laquelle ces deux processus sont mis en commun afin de pouvoir avoir un jeu complet. Nous regardons également tout ce qui a à voir avec la physique, des algorithmes de jeu, et aussi la façon de laquelle les matrices de projection et de rotation sont générées.

5.2 Initialisation du jeu

Figure 16: Déroulement de l'initialisation du jeu



Ci dessus est un diagramme représentant toutes les différentes étapes de l'initialisation du jeu, sachant que celles-ci ne coïncident pas forcément toutes avec des fonctions une à une (une étape comme schématisée peut s'étendre sur plusieurs fonctions comme elle peut ne que prendre quelques lignes dans une seul fonction. Ce sont les étapes *logiques* qui sont représentées. Le seul fichier qui ne sera pas traîté ici sera le world_manager.py puisque celui sera traîté extensivement dans le déroulement du jeu (5.3), ainsi que le terrain_generation.main.py, qui lui a déjà été traîté dans la section 3.3.1.

5.2.1 main.py

Fichier principal du jeu, qui doit être lancé pour executer le jeu. D'abord, importe la classe Game (depuis game_manager), responsable de l'initialisation du monde, puis crée son instance. Ensuite, initialise le jeu en appellant game.starting_input(), qui va demander au joueur si il souhaite créer/charger un monde, et il va faire le nécessaire pour cela. Recupère et applique ensuite les préferences de preferences.txt. Ensuite crée l'instance de la classe du monde par défaut 'World()', qui sera initialisée ainsi. Donne dans un second temps l'instance 'world' à la classe de contexte 'game', puis récupère les paramètres du monde chargé. Enfin, rentre dans le contexte du jeu (dans lequel est executée la loop principale), ce qui revient à appeler la fonction __enter__() de game, avec "with game: ". Puis lance le jeu dans le contexte avec world.run(). Nous avons créé un contexte car cela a beaucoup d'avantage. Premièrement, cela veut dire que le jeu est executé dans un environnement fermé, et que ainsi, ce qui se passe dans le contexte va être connu par le contexte. C'est très utile, car si il y a une erreur ou une interruption Ctrl-C dans le jeu, le contexte va le savoir et on va sortir du contexte de jeu Cette sortie va appeler __exit__() de game, qui va sauvegarder l'état actuel du monde. De cette manière, le contexte permet de tout controler et d'executer une fonction lorsque le jeu est quitté.

5.2.2 game_manager.py

classe Game Classe résponsable de tout ce qui est autour du jeu, à savoir la création ou le load de mondes, mais aussi la sauvegarde à la fin du jeu.

__init__ (self)
Initie des variables

__enter__ (self)
Magic Method servant à autoriser la création d'un contexte avec cette classe. Ne fait rien dans la fonction.

__exit__ (self , exc_type, exc_val , exc_tb)
Magic Method appelée lors de la sortie du contexte, donc quand le programme s'arrête, même si c'est par une erreur, et qui permet de sauvegarder notre monde quand même. Les paramètres exc_type, exc _val, exc_tb sont remplis par Python lui même, mais nous ne les utilisons pas. Cette fonction sauvegarde le monde avec self.save_world(), puis sauvegarde les chunks modifiées, puis termine et kill le process en cours (qui est le process attribué à la generation de terrain).

bind_context_to_world(self , world)
Passe l'instance du World ici, pour que cette classe y ait accès

load_paramters(self)
Load tous les paramètres qu'avait le player avant de quitter le monde, et change les paramètres du player et de la camera du world.

save_world(self)
Sauvegarde les paramètres du monde. D'abord, recupère la position, la direction et la vitesse du joueur. Load ensuite les paramètres déjà existants pour d'autres mondes. Puis ajoute les paramètres de CE monde. Sauvegarde enfin les données completes.

input_load(self)
Le monde est defini comme quoi il sera loadé, definis quel monde il faut loader. D'abord, demande le nom du monde à loader. Verifie ensuite que celui-ci existe ainsi que son dossier. Si c'est le cas, load le monde en appellant self .load_world(world_to_open).

load_world(self , world_to_open)
world_to_open : str, nom du monde à load
D'abord, intialise les settings avec ce nom de monde. Puis recupère les paramètres du monde à load, puis les met dans une variable de la classe. Ces paramètres seront appliqués seulement après que la classe World() soit instanciée. Le main.py appellera la fonction load_parameters() de cette classe.

input_create(self)
Le monde sera créé, definis quel sera le nom du world, et quelle sera sa seed. D'abord, demande à l'utilsateur un nom de monde à créer, puis si oui ou non il veut une seed customisée. Dans ce cas, demande la seed, sinon crée une seed aléatoire. Enfin crée le monde avec un nom donné et une seed donnée en appellant self .create_world(name, seed).

create_world(name, seed)
name : str, nom du monde seed : str, seed du monde Crée un monde qui s'appelle en theorie 'name' et qui a pour seed 'seed'. D'abord, load les données des mondes, puis trouve un nom unique qui n'existe pas déjà, en ajoutant "_n" à la fin du nom proposé si il existe déjà. Ensuite inite le monde dans les données de monde, initialise les settings pour ce world id, puis crée les dossiers nécessaires au world avec self.create_world_directories(). Sauvegarde ensuite les donées de monde, puis les settings. Enfin, genère le terrain initial en appellant self.generate_initial_terrain()

create_world_directories (self)
Crée les dossiers nécessaires pour le monde, à savoir le dossier de base du monde, le dossier /_outofbounds, le dossier /_outofbounds/villages, le dossier /_outofbounds/volcanos, le dossier /_entities,

create_world_directories (self)
Génère le terrain inital, ce qui va permettre d'avoir une base de terrain déjà générée, et ici avec toutes les ressources disponibles car le jeu ne tourne pas. On importe d'abord le terrain_generation.main (on le fait là, car on doit l'importer APRES l'initialisation des settings), puis on genère le terrain sur une grande zone, de 'initial_generation_size' chunks de côté, en appelant generate_initial_terrain. Ensuite, obtient les renderables pour ces chunks générés.

5.2.3 settings.py

Ce fichier est un fichier dans lequel sont trouvées la majorité des constantes immuables du jeu. Les constantes utilisateur sont elles dans les preferences.py. Ici sont définies des constantes utilisées pour le jeu, tout comme pour la génération ou encore l'affichage.

Lors de son instantiation, le fichier cherche les settings d'un monde pré-éxistant que l'on cherche à load, si on ne cherche pas plutôt à créer de monde

def setSeeds(MAIN_SEED, num_seeds, length=settings.SEED_LENGTH)
Cette fonction va calculer les sous-seeds utilisées pour différentes parties de le génération

param: MAIN_SEED
La SEED principale à partir de laquelle seront générées les sous-seeds

param: num_seed
Le nombre de de sous-seeds à générer

param: length
la longueur des sous seeds (défaut = settings.SEED_LENGTH = 16)

def init_settings (new_world_id):
Cette fonction va fixer les différentes constantes pour les formats des paths vers les différents fichiers du jeu, tels que CHUNKS_FORMAT afin de pouvoir accéder aisément à des fichiers du jeu en faisant par exemple renderables_path = RENDERABLES_FORMAT%chunk

param: new_world_id
Ceci est le "nom" du monde, essentiellement le nom de son fichier mère (Par exemple New_World_7 ou Mon_Monde). Il est utilisé pour être concaténé et forme alors les CHUNKS_FORMAT, RENDERABLES_FORMAT et HEIGHTS_FORMAT

5.2.4 preferences.py

Ce fichier va prendre le fichier preferences.txt et récupérer ses paramètres pour qu'ils puissent être utilisés dans le jeu. C'est donc un simple algorithme de parsing qui se trouve ici.

def parse_preferences ():
Cette fonction va parser les préférences et les extraire du txt. Elle est appellée par setup_preferences () L'algorithme va lire ligne par ligne et prendre tout ce qui se trouve avant un ":", le dénommer comme clé, et tout ce qui est après et le noter comme valeur. Il va ensuite prendre toutes les paires (clé: valeur) et en former un dictionnaire.
Si l'algorithme rencontre un #, il va ignorer tout ce qui suit ce dernier. Ceci permet de mettre des commentaires dans le fichier de préférences.
Si le fichier de préférences n'existe pas encore, cette fonction va le créer et y mettre le bon contenu.

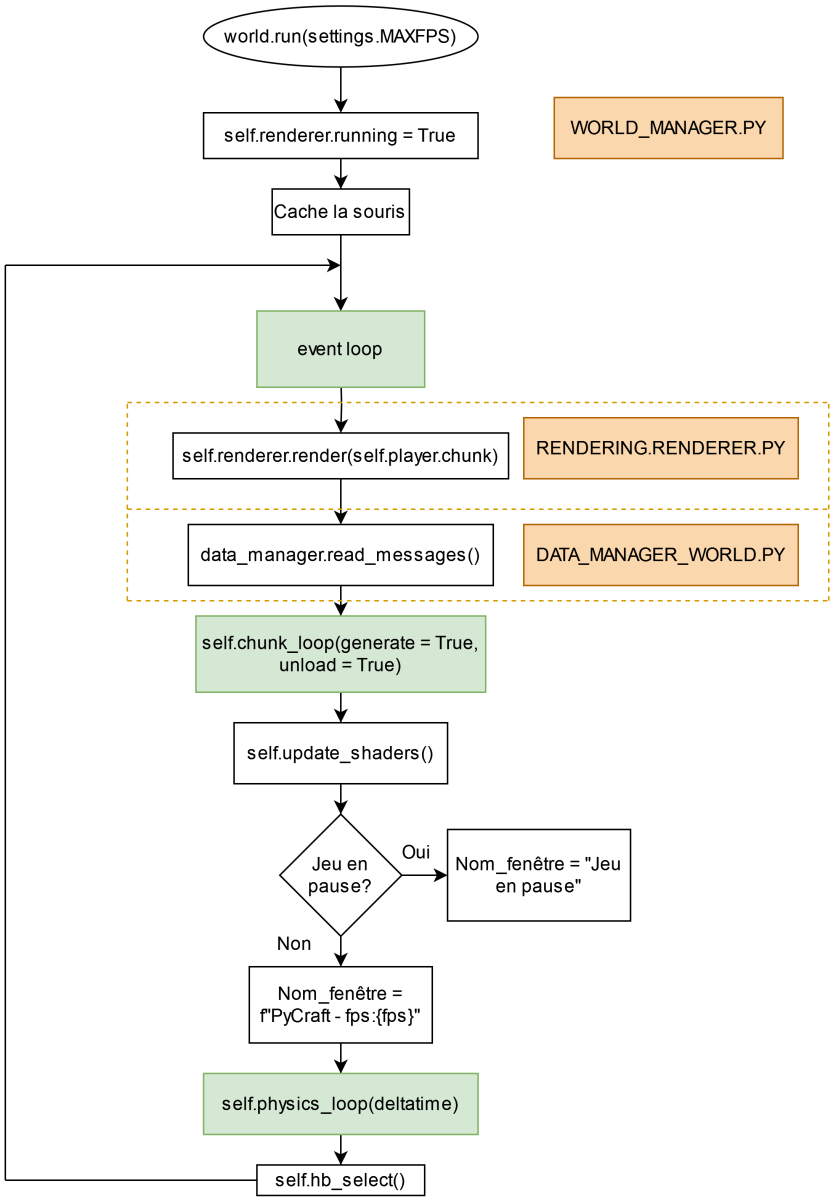
def setup_preferences ():
Cette fonction va appeler parse_preferences () et va du dictionnaire extraire les clés qui nous intéressent, vérifier que les valeurs y correspondant sont valides (si un paramètre doit être contenu entre 1 et 5 , -1 ne sera pas autorisé).
Le programme va ensuite si possible mettre les constantes aux valeurs des préférences, et sinon mettra ces dernières aux valeurs par défaut (il fera de même si une clé n'est tout simplement pas dans les paramètres). Ceci permet par exemple d'avoir le choix entre un clavier qwerty, azerty, ou autre avec des touches changeables.
Il y également les paramètres de fov, max fps, sensitivité de la souris, gravité dynamique et tous les paramètres de génération tels que le relief, les arbres, les biomes ou encore les volcans et villages.
Au sein de cette fonction en sont définies trois autres:
-get_param_pg_key qui est une fonction qui va pour un paramètre censé être une lettre vérifier si cela est le cas, et renvoyer une valeur en fonction
-get_param_value_number va faire de même que get_param_pg_key, mais pour les nombres (il y a un

paramètre pour int ou float)
-get_param_value_bool va également faire de même mais pour les bool

5.3 Déroulement du jeu: Un frame typique

Ici nous allons regarder comment un frame du jeu se déroule typiquement, du côté du jeu. Nous n'allons donc pas regarder le méchanisme de génération ou d'affichage. Leurs fonctions seront cependant citées. Les data managers ne seront pas couverts dans cette sous section non plus. Nous regarderons donc ici exclusivement le fichier world_manager. (À noter que le world_manager.World.__init__ est une fonction qui a lieu dans l'initialisation, même si nous la mettons ici)

Figure 17: Déroulement d'un frame typique du jeu



Ici les cases vertes sont les fonctions qui seront particulièrement approfondies, le reste (mis à part des parties dans d'autres fichiers) étant assez explicite. Il est cependant interessant de regarder le data_manager.read_messages(). Ce que cette fonction fait est essentiellement lire les messages venus

du processus de génération, mais nous ne le faisons qu'une fois par frame, puisque connection.poll et connection.recv sont des fonctions assez chronophages (allez voir read_messages(), 5.5.1)

5.3.1 classe: world_manager.World

Cette classe est la classe qui gère le jeu dans son ensemble, qui met en place un renderer, un joueur, des chunks, et qui génère le terrain en passant par les data_manager

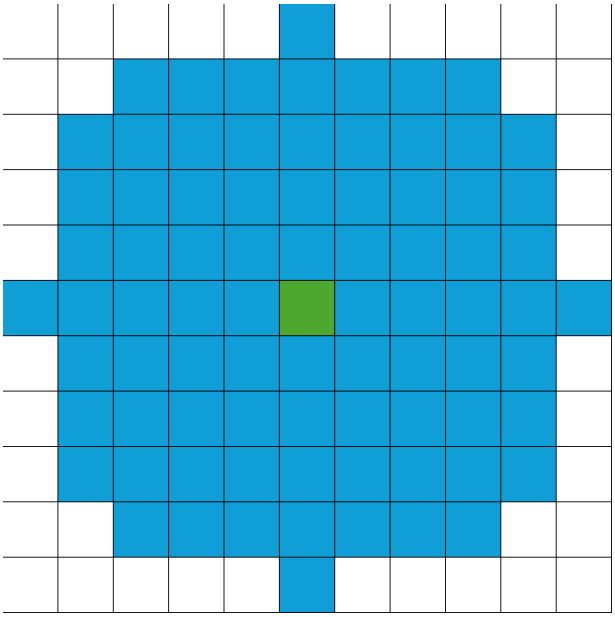
def __init__ (self):
L'initialisation d'une instance World a plusieurs phases:

Mise en place de variables: Ici sont fixées les variables self.renderdistance et self.gendistance, et sont initialisées d'autres variables telles que self.running et self.flying.
Ensuite on met en place le renderer, la scène, et finalement on fait self.renderer.bind_scene(self.scene)
Puis, on récupère le centre de la fenètre de jeu actuelle, afin de plus tard pouvoir y poster la souris.
Ensuite on met en place le joueur, ainsi que la caméra, avant d'initialiser des variables chunk, de mettre en place les texture_atlas, et de créer les shaders.
Finalement on update les uniforms des shaders récemment crées et on initialise le GUI pour le jeu

@property: renderdistance
On crée une property afin de pouvoir changer quelques variables si renderdistance venait à être changée (oui il fut un temps où nous pensions avoir le temps de mettre un menu *dans le jeu*)

Les variables que nous définissons dans la propriété sont bien évidemment self._renderdistance mais également self.relative_chunks_visible. Cette variable est une liste de tous les chunks d'une distance inférieure à renderdistance, en considérant que le joueur est à la chunk (0,0) (d'où *relative*_chunks_visible). Prégénérer cette liste permet de nous épargner beaucoup de calculs coûteux de distance à chaque frame. La liste ressemble alors, avec les carrés représentant des chunks, le carré vert au centre celui du joueur et avec une renderdistance de 5 à:

Figure 18: Les chunks "relativement" visibles aka World. relative_chunks_visible , avec une renderdistance de 5

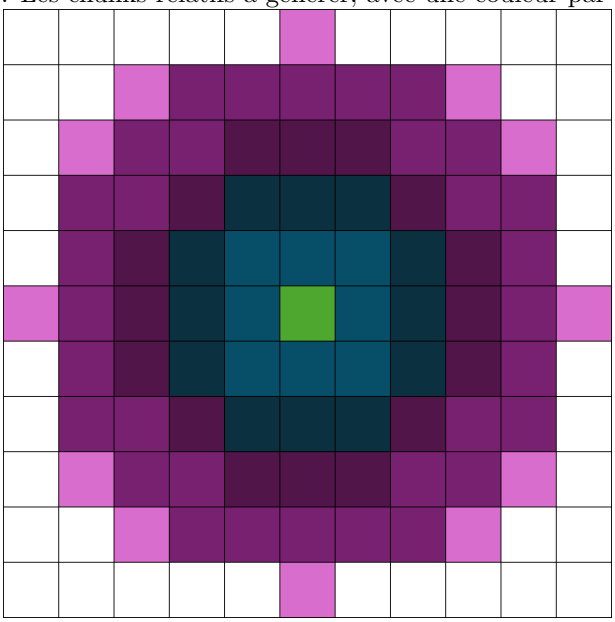


@property: gendistance

Pour les mêmes raisons que pour self.renderdistance, self.gendistance ou self.gen(eration)distance est une propriété. Cependant ici, l'algorithme utilisé pour déterminer les chunks relativement *générables* est un peu plus complexe (c'est le même au début pour générer la liste des chunks suffisamment proches) étant donné qu'en fonction de l'ordre dans lequel certains chunks seront envoyés ils seront favorisés (plus spécifiquement il est avantageux pour un chunk d'être en premier dans self.relative_chunks_to_generate. Ainsi l'ordre est important.

Donc, afin de pouvoir prioriser les chunks centraux, nous subdivisons les chunks en distance de génération en "anneaux". Ici la chunk verte représente la chunk du joueur, qui est la chunk (0,0) dans la liste (self.*relative*_chunks_to_generate)

Figure 19: Les chunks relatifs à générer, avec une couleur par "anneau"



Si vous voulez voir l'implémentation, allez voir le code dans world.world_manager.

def update_shaders(self):
Cette fonction met à jour les uniforms de caméra pour les shaders de blocs. Plus précisément, sont mis à jour la position de la caméra, la matrice de projection, et les matrices de rotation. La matrice de projection, par contre, n'est mise à jour que si elle a changé (la fenêtre a changé de taille, ce qui est un évènement assez rare). On ne fait pas pareil pour les autres uniforms de caméra puisque ceux-ci changent constamment(position et direction) À noter que les shaders du GUI ne sont pas mis à jour ici.

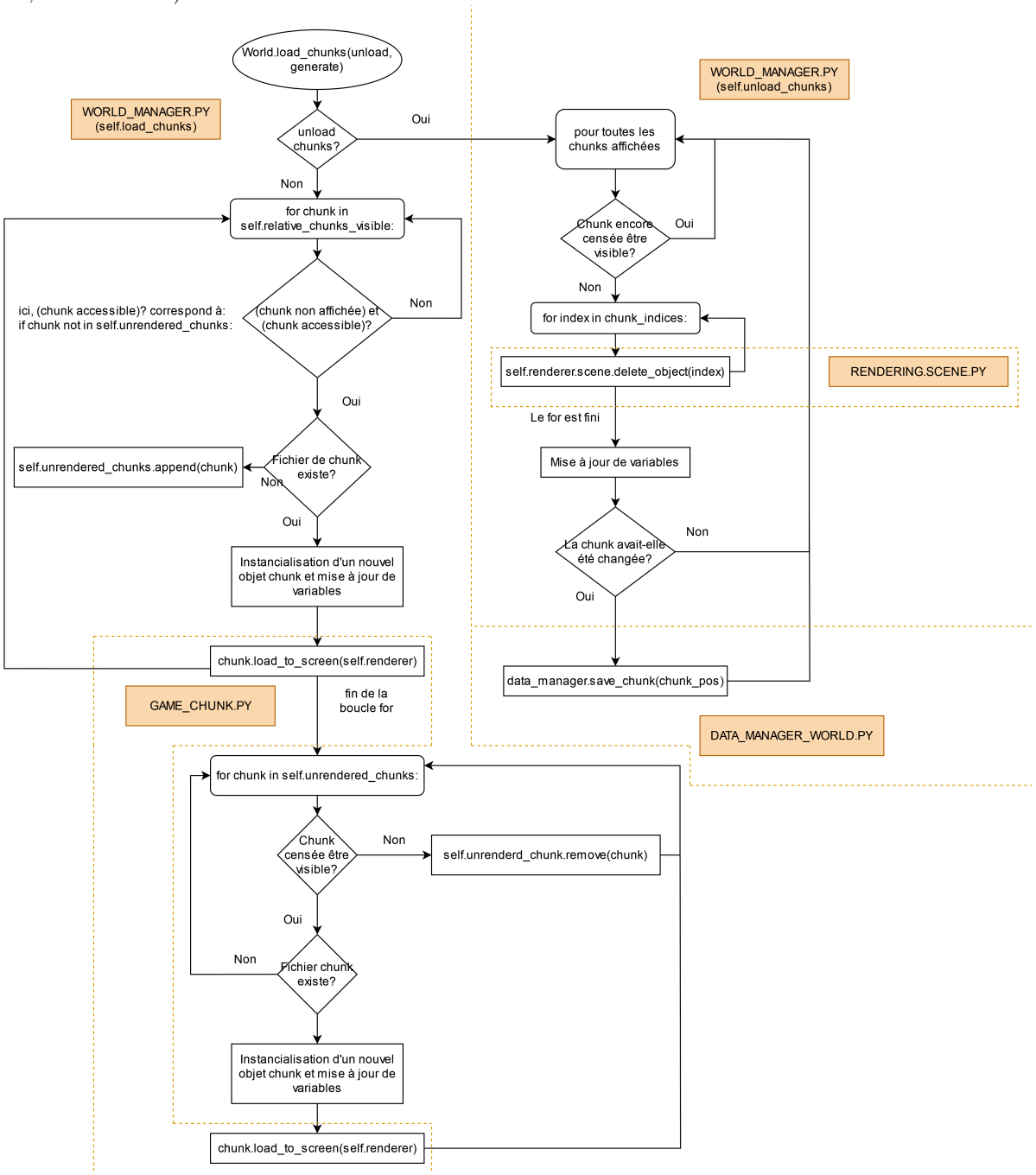
def load_chunks(self, unload=True, generate=True):
Cette fonction va prendre les chunks en distance d'affichage et les afficher si cela peut être fait. Cette fonction va également demander à ce que des chunks non générés mais normalement visibles soient générés le plus vite possible (il va les demander à la priority queue du data manager). Cette fonction est appelée par World.chunk_loop() (à 5.3.1)

param: unload
Par défaut ce paramètre est mis à True. Quand activé, le unload (désaffichage) de chunks sera effectué en faisant self.unload_chunks()

param: generate

Par défaut ce paramètre est également mis à True. Quand activé, des chunks censés être visibles, pas encore générés ou sans renderables seront redemandés à être générés (ceci passe par l'option generate du data_manager.get_renderables(chunk, *generate*, read)

Figure 20: Schéma des étapes logiques des fonctions World.load_chunks et World.unload_chunks (à load: 5.3.1, unload: 5.3.1)



Les variables qui ici pourraient paraître étranges sont self.relative_chunks_visible, et self.unrendered_chunks.

La première est définie à 5.3.1, et la deuxième est un ensemble, dans lequel sont stockées toutes les chunks que nous n'avons pas réussi à récupérer (génération trop lente, chunk avec structure, etc.) Cet ensemble va se faire itérer dessus à chaque appel de self.load_chunks, afin de voir si certaines des chunks sont encore visibles, et essayer de les récupérer de nouveau, en les générant si cela est autorisé (avec le paramètre generate, la condition en bas du schéma ci-dessus de "Fichier chunk existe" fait en réalité un appel au data_manager_world qui génèrera ou non la chunk demandée si celle-ci n'existe pas (allez voir le code)

def structure_scan(self):
Cette fonction va déterminer un scan de structures à faire et le demander au data_manager_world (le format des overchunks envoyés au data_manager est de (top_left, bottom_right)).
Un tel "scan" de structures va essentiellement itérer sur un grand nombre de chunks éloignés et déterminer si ceux-ci ont des structures (on ne peut pas générer un volcan si le joueur est juste à côté par exemple, puisque la structure pourrait s'étendre sur des chunks déjà existants)
Celui-ci nécéssite un "overchunk" soit un grand chunk de 512x512 blocs (settings.OVERCHUNK_SIZE *2). Cependant, le scan s'effectue du côté de la génération avec des grands chunks de côté settings.STRUCTURES_STEP_CHECK = 64 ainsi nous faisons en sorte de n'avoir que des coordonnées multiples de settings.STRUCTURES_STEP_CHECK. en faisant: coord//step*step
Cette fonction est appellée à chaque frame par World.chunk_loop à 5.3.1

def unload_chunks(self):
Cette fonction désaffiche les chunks hors distance de vue (self.renderdistance). Elle est appellée par self .load_chunks() quand le paramètre unload est mis à True.
Il y a un schéma représentant les étapes ci-dessous à 5.3.1
Voici donc les étapes: on itère sur tous les chunks affichés, on vérifie si une chunk est censée être visible ou non et dans le premier cas on la désaffiche. La raison pour laquelle il y a une boucle, est car il peut y avoir plusieurs chunkbatchs, soit objets chunks dans la vram par chunk. Ainsi il nous faut tous les supprimer. Puis nous mettons à jour quelques variables de chunks (chunks affichés, positions de chunks affichés etc.) Avant de terminer nous regardons si la chunk avait été modifiée, et si elle l'a été, nous la sauvegardons en utilisant le data_manager_world.save_chunk(chunk)

def gen_chunks(self):
Cette fonction va demander au data_manager_world de générer toutes les chunks non générées.
Elle va donc itérer sur la liste self.relative_chunks_to_generate (voir 5.3.1) et demander à générer tout chunk non généré. Ceci est fait en ajoutant chaque chunk à une liste, et en donnant la liste comme batch à data_manager_world.send_batch() (à 5.5.1)ce qui est plus rapide pour beaucoup de chunks.
Elle est appellée par World.chunk_loop à 5.3.1

def chunk_loop(self, unload=True, generate=True):
Ceci est la fonction qui sera appellée par World.run_loop() (à 5.3.1) Elle appellera la fonctions World.load_chunks(unload, generate) (à 5.3.1) ainsi que les fonctions World.gen_chunks() à 5.3.1 et World.structure_scan() à 5.3.1.Ces dernières ne seront cependant appelées que si generate == True

param: unload=True
Si le code doit désafficher des chunks (utile pour le débuggage)

param: generate=True
Si le code doit générer de nouveaux chunks (on pourrait le désactiver par préférence personnelle)

def click (self , event):
Cette fonction est appellée lors d'un clic dans la event loop (5.3.1) Si un clic gauche est effectué, on appelle World.break_block() à 5.3.1
Si un clic droit est effectué, on appelle alors World.place_block à 5.3.1

param: event
Ce paramètre est fourni par pygame et contient les informations du clic, et en ce qui nous intéresse les informations nous disant si cela est un clic gauche ou droit

def break_block(self):
Fonction appellée lors d'un clic gauche par World.click() 5.3.1. Cette fonction va casser le bloc dans la ligne de vue du joueur.
Elle va donc d'abord voir si le bloc regardé par le joueur (self.lookat = self.player.look_at_block(5) à 5.4.3) n'est pas inexistant
Ensuite le programme modifie la chunk stockée en ram chez le data_manager_world (il casse le bloc). Puis est appellée la fonction de terrain_generation.fast_renderables.cassage() à 3.3.8.
Ce que cette fonction fait est de renvoyer une liste de blocs nouvellement visibles (quand nous creusons, la terre sous l'herbe n'est pas visible avant que nous ne cassions celle-ci). Ainsi on itère sur les "changes" soit ces blocs nouvellement visibles, et on ajoute chaque nouveau bloc au game_chunk dans lequel il appartient en utilisant game_chunk.add_block(block) à 5.4.5.
On supprime finalement le bloc initial dans son game_chunk avec game_chunk.delete_block à 5.4.5

def place_block(self):
Fonction appellée lors d'un clic droit par World.click() à 5.3.1. Cette fonction va placer le bloc sélectionné

par l'utilisateur dans la hotbar dans la ligne de vue du joueur.
Nous regardons donc d'abord si le bloc regardé par le joueur (self.lookat = self.player.look_at_block(5) à 5.4.3) n'est pas inexistant.
Ensuite, on match case self.lookat[3], soit la direction par laquelle la ligne de vue du joueur a en dernier traversé le bloc regardé (de la gauche, de la droite, de devant, d'en dessous, etc.)
Une fois que nous avons récupéré le bloc à placer, nous vérifions que le bloc que nous voulons placer ne colisionne pas avec le joueur, qu'il est contenu entre 0 et la limite verticale des blocs dans le jeu, et nous regardons aussi que le bloc n'existe pas déjà visuellement dans son game_chunk, en cas de potentiel bug de la part de l'algorithme de regardage de bloc (c'est suffisamment peu coûteux et l'évènement de placement de bloc est tellement rare que nous pouvons nous le permettre).
Nous calculons ensuite l'id du bloc que nous voulons placer, qui est celui du bloc sélectionné. Ceci est fait en récupérant la position du carré blanc de sélection de bloc ainsi que le niveau de hotbar utilisé.
Finalement nous ajoutons le bloc à son game_chunk avec game_chunk.add_block(block) à 5.4.5 et nous changeons les chunks en ram avec le data_manager_world

def hb_select (self):
Cette fonction traite les touches nécéssitées pour changer de bloc sélectionné dans la hotbar, ainsi que de changer les uniforms dans les shaders correspondants pour afficher le changement (le mouvement de carré blanc ou le niveau de hotbar utilisé)
Nous regardons donc, pour toutes les touches de hotbar si elles sont dans les touches appuyées (un ensemble), et dans le cas affirmatif nous changeons les variables self.hbcounter (la position du carré blanc) ou self.hblevel (le niveau de hotbar soit quelle image est affichée).
Ensuite nous mettons à jour des uniforms dans les shaders hbselection(le carré) et hotbar(la texture de la hotbar), et finalement nhous retirons la touche de hotbar appuyée de l'ensemble (voir 5.3.1)
Cette fonction est appellée par World.run_loop (à 5.3.1

def physics_loop(self , deltatime=0:
Cette fonction est appellée par le World.run_loop() à 5.3.1 si le jeu n'est pas en pause.
Elle va effectuer tous les changements ayant à voir avec mouvemement, rotation et bien évidemment la physique.

param:deltatime=0
Le deltatime est le temps qui s'est écoulé depuis le dernier frame du jeu. Nous en avons besoin pour pouvoir avoir une simulation plus fluide: disons que nous bougeons de 0.5 blocks en 0.1 sec, il nous faudrait donc bouger de 1 bloc en 0.2 secs en toute logique, afin de ne pas avoir un mouvement saccadé.
C'est à cela que servira donc le deltatime, et donc si vous le voyez en facteur à une multitude d'endroits, ceci est la raison

l'éxecution
Nous commençons par calculer le déplacement de la souris depuis le dernier frame, et nous remettons

cette dernière au centre de la fenêtre.
Puis nous calculons la quantité de mouvement pour les déplacements, et la limitons à 0.3 pour éviter des bugs
Puis pour chacune des directions de mouvement nous regardons si la touche est appuyée, et bougeons le joueur en fonction. Les exceptions sont le mouvement vertical, qui lui va se comporter comme les autres si self.flying == True (le joueur a activé le mode vol) ou si le joueur est dans un liquide (comportement similaire de ce côté-ci), et sinon va ignorer le DOWN_K et faire sauter le joueur sur appui du UP_K.
Ensuite on appelle la fonction du joueur player.Player.simulatePhysics 5.4.3 en passant le deltatime, et en mettant le paramètre gravité à not self.flying afin de ne l'activer que quand nous ne volons pas.
Dernièrement nous calculons à partir du mouvement de la souris l'angle par lequel tourner le joueur et la caméra, les tournons par cet angle, et appellons player.Player.look_at_block() 5.4.3 afin de regarder un bloc.

def run(self, max_fps):
Ceci est la boucle maître appellée par main à la fin de l'initialisation (voir 5.2) qui éxécutera tout le jeu mis à part la partie génération.
Il y un schéma illustrant toutes les étapes de la run_loop à 5.3

param: max_fps
La restriction sur les fps, purement par raison esthétique(il peut être préférable d'avoir un 80fps constant plutôt qu'un 110fps oscillant)

l'éxécution
Avant le la boucle infinie du jeu, nous mettons la variable "running" du World.renderer à True, afin que celui-ci puisse bien fonctionner. Ensuite, si le jeu n'est pas en pause, nous cachons la souris.
Nous commençons maintenant la boucle infinie.
On commence par récupérer le deltatime (le temps qui s'est écoulé depuis le dernier frame) avec self.renderer.clock.tick(max_
puis vient la event loop (détaillée en dessous), et on demande au renderer de réafficher les chunks, une fois avoir appelé self.update_shaders() (à 5.3.1).
Nous lisons ensuite les messages venus de la génération avec data_manager.read_messages (à 5.5.1), avant d'appeler self.chunk_loop(unload=True, generate=True) (voir 5.3.1)
Finalement, on regarde si le jeu est en pause, on met le nom de la fenêtre en fonction, et si cette proposition est vraie, on appelle également self.physics_loop (5.3.1) ainsi que self.hb_select (5.3.1)

la event loop
Ceci est la boucle incrustée dans la boucle du run_loop qui va itérer sur tous les évènements nouveaux de pygame (touches appuyées, fenêtre bougée, etc.).
Voici un schéma de la boucle

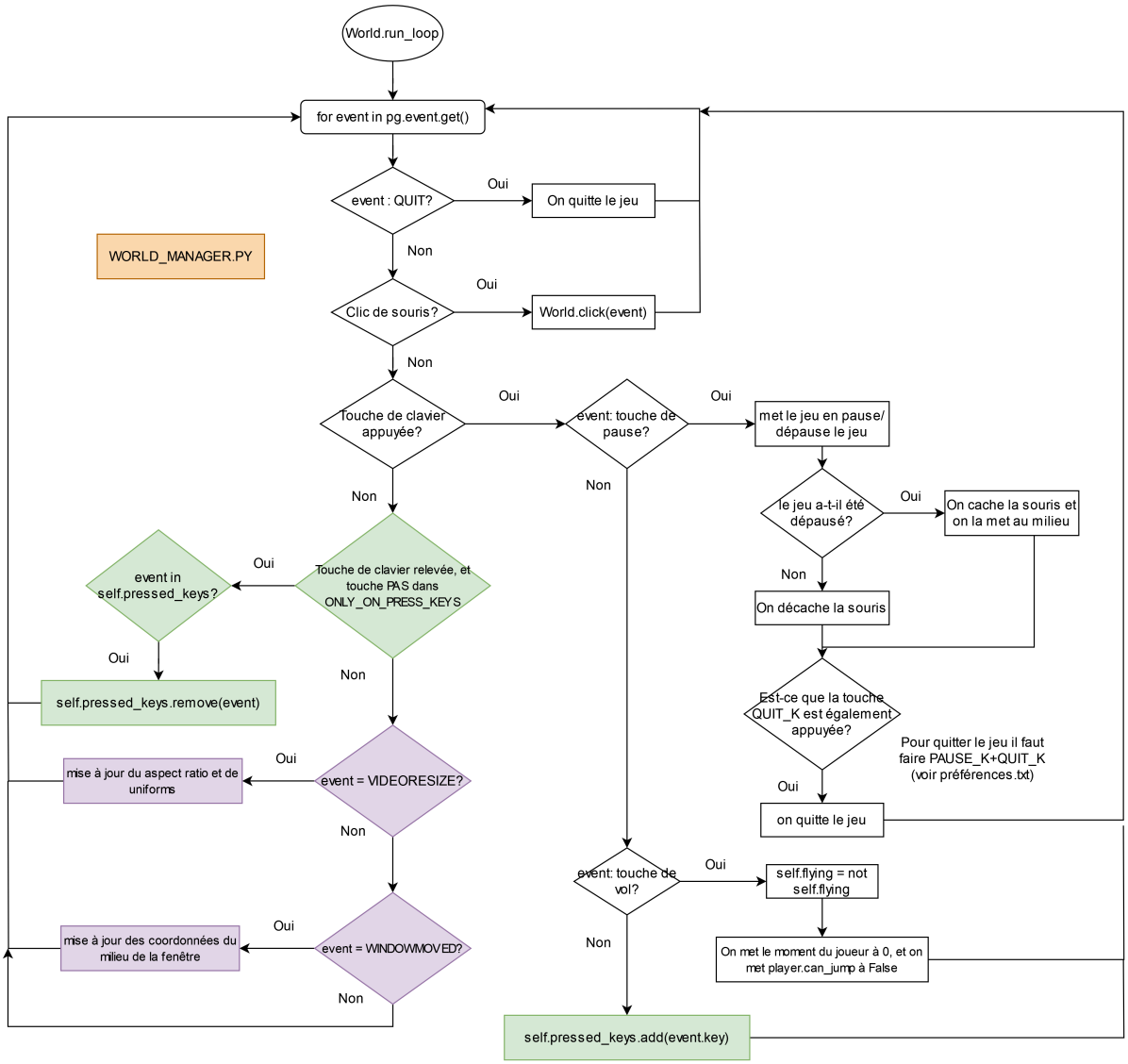


Figure 21: La event loop dans World.run_loop

Cette boucle est assez complexe, mais pas très compliquée. Les principales parties qui peuvent être étrange sont les parties schématisées en vert, et celles en violet.

Les éléments en vert, traitent tout ce qui a à voir avec l'ensemble *pressed_keys*. C'est un ensemble puisque les opérations effectués dessus sont "in", "add" et "remove", et c'est ce pour quoi un ensemble est avantageux (x in set() : O(1) tandis que x in list peut être O(n) au pire), de plus il n'y a pas de risque de duplicat.

pressed_keys est donc un ensemble auquel sont ajoutées la plupart des touches lorsqu'elles sont appuyées, et auquel sont enlevées la plupart des touches lorsqu'elles sont relâchées. Ceci permet donc de savoir quelles touches l'utilisateur est en train d'appuyer.

Il y a ici marqué la "plupart" des touches, puisque certaines touches ne sont censées être traîtées que lors de l'appui initial (touche de pause, de vol, etc.). Ces touches là ne seront donc pas ajoutées à pressed_keys, et seront au contraire traitées immédiatement(elles constituent les ONLY_ON_PRESS_KEYS).

Cependant, les touches de hotbar y sont ajoutées, mais n'en sont pas retirées. En effet, afin de diviser la event loop pour éviter d'avoir quelque chose de trop complexe, nous ajoutons les touches de hotbar à l'ensemble self.pressed_keys, et self.hb_select (à 5.3.1) va les enlever de là immédiatement après éxécution des actions (bouger le carré blanc...) afin d'obtenir le même résultat que les autres touches censées n'être traîtées que lors de l'appui initial.
Les éléments en violet, eux, ont à voir avec les évènement de fenêtre de pygame, particulièrement, pg.VIDEORESIZE est donné lorsque les dimensions de la fenêtre ont changé, dans quel cas on recalcule le aspect ratio (largeur/longueur), on met alors à jour la variable de ce même nom de la caméra du world, afin qu'elle mette à jour sa matrice de rotation, et on met à jour des uniforms dans les shaders en dépendant, soit les shaders cross, hbselection(le carré de sélection) et hotbar(la texture affichée pour la hotbar).

5.4 Les objets du jeu: Le GameObject, le Player, la Caméra et la Chunk

Nous allons maintenant nous intéresser aux objets du jeu, qui sont stockés dans les fichiers game_object.py(le GameObject), player.py(le Player et la Caméra) et game_chunk.Chunk. Ces classes seront toutes utilisées par le world_manager.py (à 5.3.1). Elles sont centrales au bon fonctionnement et à la modulabilité du jeu, même si elles peuvent paraître redondantes.

5.4.1 Repères d'espace et direction

Cependant, avant de pouvoir attaquer ces classes, il nous faut regarder comment nous nous repérons dans l'espace et comment la direction d'objets est représentée.
Voici donc un schéma représentant le repère utilisé tout au long du jeu

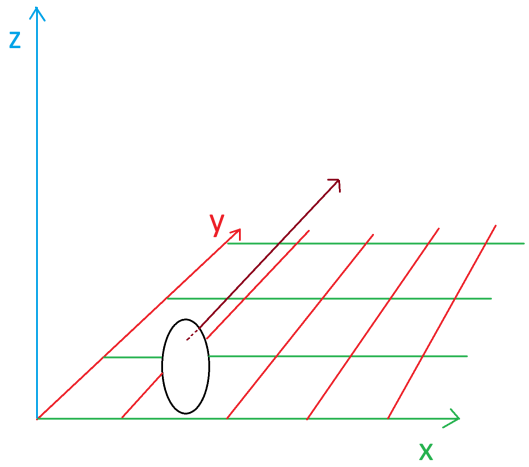


Figure 22: Repère dans le monde, avec le joueur et sa direction en rouge foncé

Ainsi un sol plat se situerait entièrement dans le plan xy, et tout relief serait un changement en z. À noter qu'ici, la direction standard est dans la direction et dans le sens de l'axe y, et donc il faut se dire que
"devant": +y
"droite": +x
"derrière": -y
"gauche": -x
Il est aussi important de savoir que l'altitude correspondant à z=0 est le fond du monde.

Regardons maintenant comment la direction d'un objet dans le jeu est stockée. Notamment, la direction est un vecteur (généralement de magnitude 1), et donc, peut être représentée avec

$$\text{direction} = \begin{bmatrix} \text{direction}_x \\ \text{direction}_y \\ \text{direction}_z \end{bmatrix}$$

Soit, illustré avec une image,

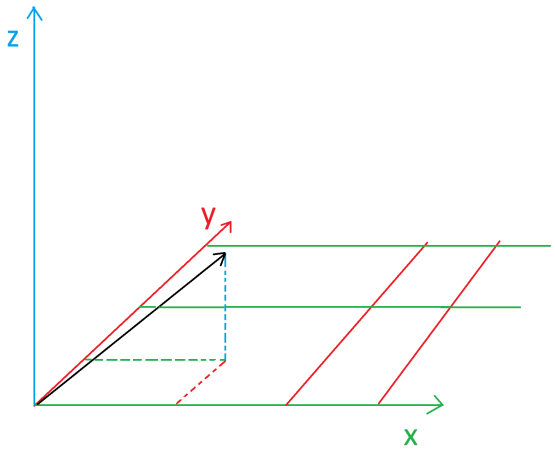


Figure 23: La direction d'un objet centré à l'origine du repère, en noir, avec ses composants selon chaque axe en pointillé

Cette représentation de la direction nous sera très utile maintes fois, mais une autre est nettement plus pratique pour tourner les objets, et c'est la représentation avec angles que voici:

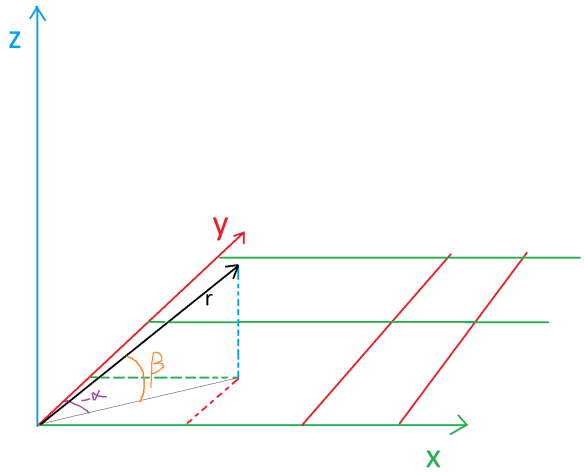


Figure 24: La direction d'un objet centré à l'origine du repère, en noir, de magnitude r, avec les angles α et β dessinés

Il est assez simple de changer entre les deux:
D'abord, la longueur de la projection du vecteur direction sur le plan xy, est de $r\cos(\beta)$. De là,

$$\text{direction} = \begin{bmatrix} r\sin(\alpha)\cos(\beta) \\ r\cos(\alpha)\cos(\beta) \\ r\sin(\beta) \end{bmatrix}$$

et pour l'autre sens, $\beta = \arcsin\left(\frac{\text{direction}_z}{\|\text{direction}\|}\right)$ et $\alpha = \arccos\left(\frac{\text{direction}_y}{\|\text{direction}\|\sin(\beta)}\right)$

5.4.2 classe: game_object.GameObject

Cette classe est la base pour tout objet dans la scène, ce qui pour l'instant constitue l'objet player et l'objet camera, ce qui peut sembler peu, mais il faut savoir qu'avoir structuré nos objets d'une telle façon nous permettrait d'introduire de nouvelles entités du jeu très aisément.
Nous aurons donc vu la motivation derrière le GameObject, voici maintenant son implémentation.

def __init__ (self , position , direction=(0, 1, 0), following_position =None, tag=None):
Ici nous initialisons le GameObject, et tout ce qui est fait est simplement une initialisation des variables en fonction des paramètres que nous allons voir. Est également effectuée à la fin de l'initialisation la fonction GameObject._updateAngles() (voir 5.4.2), afin de déjà pouvoir calculer les angles.

param: position
Ceci est la position du GameObject, un tuple (x,y,z) (voir le repère du jeu à 22)

param: direction=(0,1,0) Ceci est la direction du GameObject, en tuple (x,y,z). (0,1,0) est la direction par défaut (nous regardons devant) (voir la façon de laquelle la direction est gérée à 23)

param: following_position=None Ce paramètre est un peu plus subtil. Essentiellement il faut passer un autre GameObject, et lorsque ce dernier sera bougé, le GameObject nouvellement crée va également bouger.
Laisser à None pour ne pas suivre d'objet (ceci avait été pensé dans la lignée d'avoir une multitude de caméras différentes, dans quel cas un tel dispositif simplifie grandement la syntaxe)
À noter que dans le cas qu'un GameObject en suive en autre, la propriété @self.position.setter() ne fixera que la position *relative* de l'objet par rapport à celui qu'il suit.

param: tag=None
Ceci est un paramètre non utilisé, mais qui avait été pensé comme un identifiant lambda que nous aurions pu utiliser (par exemple pour caractérisér différents types d'entités qui seraient toutes des instances de la même classe).

@property: def position(self):
Nous faisons ici une propriété pour la position du GameObject que nous appelerons A, dans le cas que celui-ci soit en train de suivre un autre objet, disons B (self.following_position à 5.4.2), dans quel cas nous retournons la position relative de A par rapport à B plus la position de B.

@property: def direction(self):

À cause des différentes représentations de la direction (voir 23), nous faisons une propriété pour fixer la direction (en représentation vectorielle), qui mettra à jour la représentation vectorielle si les angles ont changé, et qui inversément mettra self._angles_updated à False quand la représentation vectorielle vient à être changée.

@property: def angles(self):

Cette propriété permet de récupérer la représentation angulaire de la direction de l'objet (voir 23). C'est une propriété puisque suite à un changement de direction (représentation vectorielle), les angles ont changé, et il faut donc les recalculer. C'est précisément ce que fait cette propriété en appelant GameObject._updateAngles() (5.4.2)

def _updateAngles(self):

Cette fonction est appellée par la propriété GameObject.angles (à 5.4.2) et va recalculer les angles à partir de la direction (voir 23). Le calcul effectué est le même qu'ici (5.4.1) même si c'est légèrement différent dans l'implémentation pour cause que nous gardons également les valeurs de $\sin(\alpha)$,$\cos\alpha$),$\sin(\beta)$ et $\cos(\beta)$

def turn(self , alpha_turn, beta_turn=0):

Cette fonction permet de tourner le GameObject d'un certain angle dans les deux dimensions angulaires (voir 23). Elle va donc ajouter les valeurs de alpha_turn et de beta_turn à leurs angles respectif, et calculer la représentation vectorielle de la direction (les calculs sont à 5.4.1).

param: alpha_turn

L'angle en radians à ajouter à l'angle alpha

param: beta_turn

L'angle en radians à ajouter à l'angle beta

def move(self, movement, relative=True):
Cette fonction bouge l'objet par le vecteur mouvement, et ce relativement à la direction (mis à part pour le déplacement z) quand le paramètre est activé

param: movement
Le vecteur mouvement par lequel bouger l'objet (un tuple, une liste, un array, essentiellement un itérable)

param: relative
Si le mouvement doit être relatif à la direction (disons que nous avançons vers l'avant, et que nous regardons au nord-est (+x,+y), en activant ce paramètre, dire de bouger de (0,1,0) nous bougera de 1 unité dans la direction, soit vers le nord-est. Ceci n'est cependant pas appliqué pour le mouvement selon z, puisque généralement les objets bougent normalement vers le haut et le bas peu importe où ils regardent. Les calculs pour effectuer ceci sont un calcul matriciel, puisque nous voulons essentiellement changer de base avec
$\hat{i} \rightarrow \hat{i}$tourné de α et
$\hat{j} \rightarrow \hat{j}$tourné de α sous le diagramme ci-contre, sachant que la coordonnée z ne change pas (le mouvement n'est relatif que dans le plan xy). ATTENTION, ici nous n'utilisons pas la direction stockée, mais les angles, puisque nous voulons avancer d'autant que nous regardions en l'air ou devant nous.

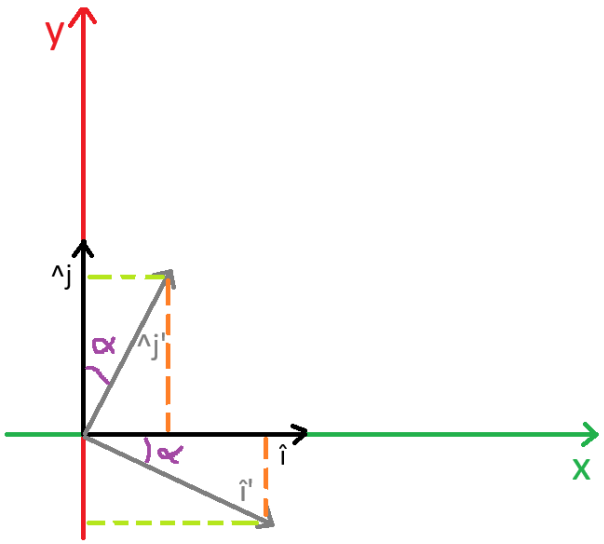


Figure 25: Les vecteurs unitaires non tournés et tournés (noirs et gris respectivement). Le mouvement relatif s'effectue avec une rotation des vecteurs unitaires par α

Ici donc, on peut voir que les coordonnées des vecteurs $\hat{i}'$ et $\hat{j}'$, soit les projections vertes et oranges, sont de

$$\hat{i}' = \begin{bmatrix} -\sin(\alpha) \\ \cos(\alpha) \end{bmatrix};$$

α est ici négatif, donc il a fallu inverser le signe de son sinus

$$\hat{h}' = \begin{bmatrix} \cos(\alpha) \\ \sin(\alpha) \end{bmatrix}$$

Mais ici pas de problème de signe

Ainsi la matrice pour effectuer le mouvement relatif devient

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

et le mouvement absolu est alors

$$\text{mouvement absolu} = \begin{bmatrix} \text{mvt}_x \\ \text{mvt}_y \\ \text{mvt}_z \end{bmatrix} \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \text{mvt}_x \cos(\alpha) - \text{mvt}_y \sin(\alpha) \\ -\text{mvt}_x \sin(\alpha) + \text{mvt}_y \cos(\alpha) \\ \text{mvt}_z \end{bmatrix}$$

5.4.3 classe: Player(GameObject)

Nous allons maintenant passer au fichier player.py, et nous intéresser à la classe Player, utilisée par le world_manager (à 5.3.1). Elle hérite de la classe GameObject (à 5.4.2), et définit l'entité du joueur.

def __init__ (self , position , direction=[0, 1, 0], following_position =None, tag=None):
Les paramètres ci-dessus sont utilisés pour initialiser le super(), soit le GameObject. Allez donc voir ce que ces paramètres font là bas (à 5.4.2
Il y a cependant trois nouvelle variables définies dans le init du player, qui sont
can_jump: le joueur peut-il sauter?
in_water: le joueur est-il dans l'eau (ou dans un autre fluide)? speed_y: la vitesse du joueur verticalement

@property: def position(self):
On écrase la propriété du même nom de GameObject (5.4.2), afin d'y incorporer le calcul de la chunk du joueur, qui est très utilisée dans le world_manager. La chunk n'a pas été incorporée dans le GameObject directement, puisque nous n'en avons que besoin pour le joueur. Le fonctionnement de la propriété mis à part cela est le même.

def simulatePhysics(self , term_velocity=−0.9, deltatime=0, gravity=True):
Cette fonction va effectuer les étapes de la simulation de la physique du joueur, donc appliquer la gravité, et éviter que le joueur ne traverse des blocs. Elle est appellée par world_manager.World.physics_loop

param: term_velocity = -0.9
Ce paramètre détermine la vitesse terminale, soit la vitesse maximum par laquelle le joueur peut descendre en un frame. Ceci peut être mis à False pour aucune, même si cela est fortement déconseillé. En effet, la vitesse terminale est mise à un nombre qui fait en sorte que le joueur ne puisse jamais traverser un bloc même en tombant à sa vitesse maximale (imaginez que l'on descende tellement vite que le joueur se téléporte en dessous d'un bloc et que donc nous ayions traversé un bloc).

param: deltatime=0
Le deltatime est le temps qui s'est écoulé depuis le dernier frame du jeu. Nous en avons besoin pour pouvoir avoir une simulation plus fluide: disons que nous bougeons de 0.5 blocks en 0.1 sec, il nous faudrait donc bouger de 1 bloc en 0.2 secs en toute logique, afin de ne pas avoir un mouvement saccadé.

param: gravity=True
Si la gravité est activée. Utilisé pour la méchanique de vol du joueur

Initialisation et gravité

Au début nous mettons les variables self.in_water et self.can_jump à False. Puis, si la gravité est activée, nous regardons si le joueur est en dessous de la borne minimale de GRAVITY_THRESHOLD ou si la gravité dynamique n'est pas activée (GRAVITY_THRESHOLD est une constante de settings utilisée pour la gravité dynamique, soit que le plus haut que nous montions, le moins la gravité ne s'applique, avant d'éventuellement s'inverser).

Dans ce cas nous appliquons donc la gravité de façon normale, et dans l'autre nous rajoutons un facteur afin de faire fonctionner la gravité dynamique. Finalement, nous limitons la vitesse maintenant accélérée par la gravité à sa vitesse terminale. Puis nous bougeons le joueur (à nouveau cap à -0.9z). Également, nous faisons en sorte d'éviter au joueur de s'envoler trop haut ou trop bas (entre -10 et 1000). Ceci est alors la fin de cette étape

Traîtement des collisions

Nous créons ici au début des tuples de toutes les positions de blocs avec lesquelles le joueur peut entrer en collision, organisées en différents groupes. Nous les avons ci-dessous représentés avec différents blocs, basalte pour le groupe du sol, planche pour le plafond, les différents minerais pour les 4 groupes de collision de côté et le sable pour les groupes de collision de diagonale (unee colonne par groupe). Il y également le groupe de collisions dans le joueur, soit les blocs de neige et de lave, qui représentent également le joueur, même si ils n'y sont pas toujours les deux affichés pour des raisons de visibilité. Ainsi

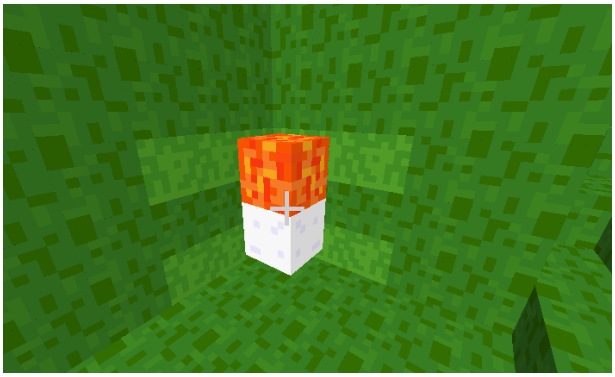


Figure 26: Représentation du joueur par les blocs de lave et de neige (le premier est la tête)

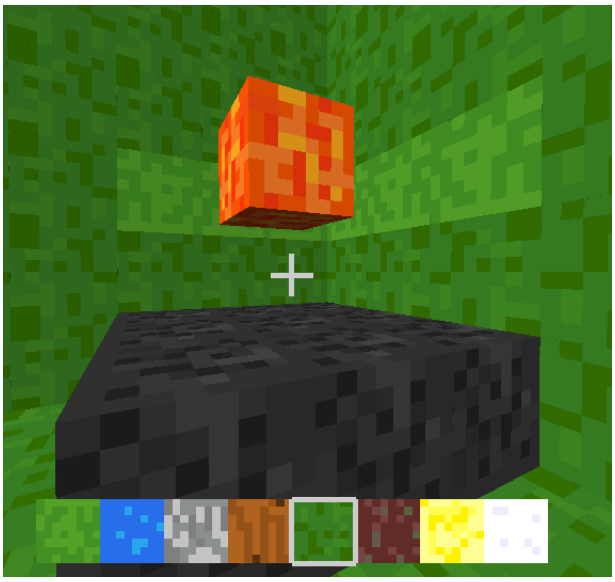


Figure 27: Représentation des blocs de collision de "sol", avec la lave représentant la tête du joueur, à un bloc au dessus du centre du carré de basalte

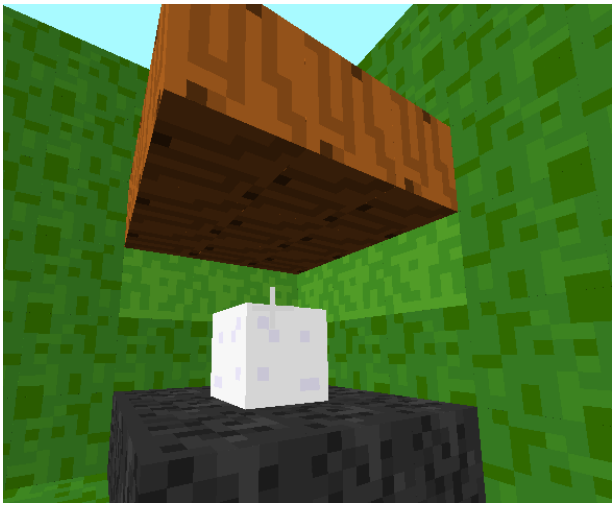


Figure 28: Représentation des blocs de collision de "ciel", avec la neige représentant les pieds du joueur, à un bloc en dessous du centre du carré de planches

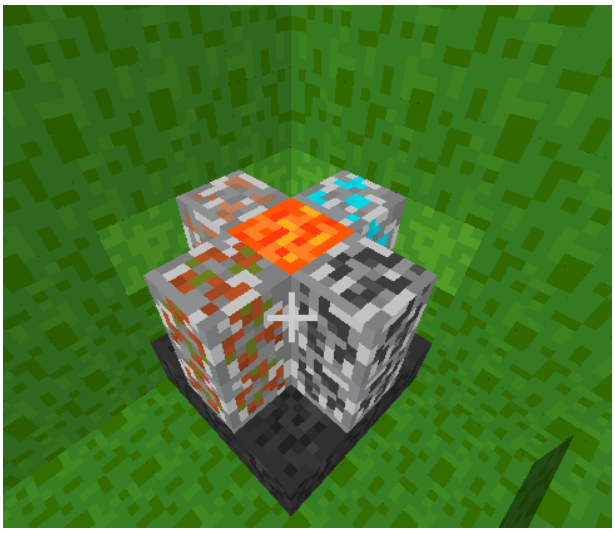


Figure 29: Représentation des blocs de collision de "côté", avec la lave représentant la tête du joueur, et chaque minerai représentant un groupe de collision différent

Figure 30: Représentation des blocs de collision de "diagonale", avec la lave représentant la tête du joueur, la neige ses pieds et chaque colonne de sable un groupe de collision différent

L'algorithme pour traîter les collisions itère alors sur toutes ces collisions, et alors, en cas de collision (voir 5.4.3), en fonction du type de collision, on remonte le joueur (cas de collision intérieure, ou avec le sol(on met alors Player.can_jump = True)), ou on peut le descendre (cas de collision avec le ciel), ou encore on remet le joueur au bon endroit si il collisionne avec les côtés ou les diagonales (pour ces dernières, on se demande de quel côté décaler, et cela se fait dans la direction où il faut le moins se décaler (ça peut être x tout comme y en fonction d'où on approche). Ces mécaniques changent cependant si le joueur se trouve dans de l'eau, dans quel cas on divise tout simplement la vitesse verticale du joueur par 2, ou si le

joueur est sur de l'eau, dans quel cas nous nous enfuyons en sautant

def colliding (self , coordinates_1 , coordinates_2):
Cette fonction va vérifier si le joueur collisionne avec le cube de sommets opposés coordinates_1 et coordinates_2 comme ci-dessous

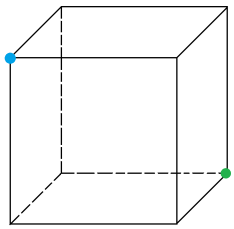


Figure 31: La hitbox du cube avec lequel nous effectuons un test de collision

Pour ce faire, la fonction établit la hitbox du joueur, avec des paramètres établits dans settings, et vérifie pour certains points de la hitbox si ils sont dans celle du cube(on regarde si le point est contenu dans le cube dans la direction x, puis y, puis z). Si oui, True est return.
Comment trouve-t-on ces "certains" points? Voici un schéma que nous allons utiliser pour y répondre

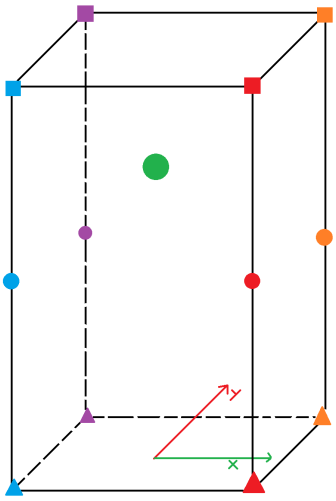


Figure 32: La hitbox du joueur, avec un point/carré/triangle par point que nous vérifions, et le point vert comme position du joueur (centré dans les xy, mais dans la partie supérieure de la hitbox)

Nous allons illustrer le problème, lors d'une collision avec un bloc, en fonction de où ce dernier est il serait inutile de vérifier pour *tous* les points du la hitbox.

Imaginez une collision entre le joueur et le sol, il serait inutile de vérifier pour le point schématisé par un carré jaune, ainsi, nous avons en fonction du type de collision, un type différent de check de collision, avec que quelques points de la hitbox inclus. Voici donc ces types de collisions, et les points de la hitbox du joueur pris en compte:
0: Collision de sol, nous prenons en compte les triangles, et les cercles (si nous descendons trop vite ceux-ci sont nécessaires pour éviter de traverser des blocs)
1: Collision de plafond, nous prenons en compte les carrés, et les cercles (si nous montons trop vite ceux-ci sont nécessaires pour éviter de traverser des blocs)
2: Collision de côté gauche, nous prenons en compte l'arète violette ainsi que la bleue
3: Collision de devant, nous prenons en compte l'arète violette ainsi que l'orange
4: Collision de droite, nous prenons en compte l'arète orange ainsi que la rouge
5: Collision de derrière, nous prenons en compte l'arète rouge ainsi que la bleue
6: Collision de diagonale devant-gauche, nous ne prenons en compte que l'arète violette
7: Collision de diagonale devant-droite, nous ne prenons en compte que l'arète orange
8: Collision de diagonale derrière-gauche, nous ne prenons en compte que l'arète rouge
9: Collision de diagonale derrière-droite, nous ne prenons en compte que l'arète bleue
10 : Ce cas-ci est utilisé pour le placement de blocs, et prend en compte tous les points

param: coordinates_1
Ceci est un tuple de coordonnées représentant un des sommets diagonalement opposés du cube

param: coordinates_2
Ceci est l'autre tuple de coordonnées représentant l'autre sommet diagonalement opposé du cube

def look_at_block(self ,max_reach=5):
Ceci est la fonction qui va permettre de déterminer quel bloc le joueur regarde.

param: max_reach
Ceci est la distance maximum à laquelle l'algorithme va regarder (distance qui peut varier si la direction n'est pas unitaire)

L'algorithme
Afin de simplifier les choses, pour comprendre l'algorithme, nous nous restreindrons à 2 dimensions. Tout peut toutefois être facilement étendu en 3D. Le problème est alors, qu'il nous faut en quelque sorte marcher sur la ligne de vue du joueur (sa direction, voir 23), pas à pas et regarder dans quel bloc nous sommes 1a chaque fois, puis regarder quel type de bloc ce dernier est, et dans le cas de l'air nous continuons. Cette méthode est schématisée ci-dessous

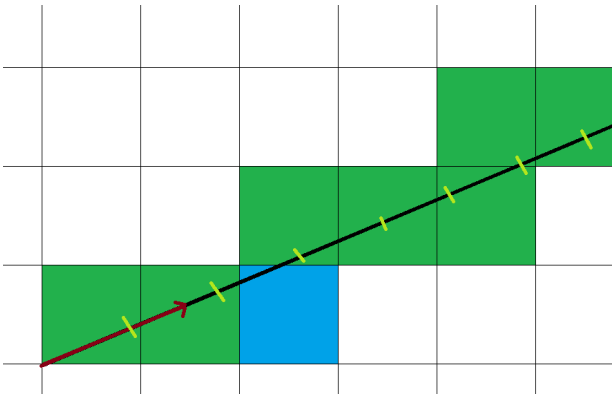


Figure 33: Schéma de la première méthode envisagée pour regarder un bloc, avec la demidroite noire représentant la continuation du vecteur direction(rouge foncé). Les traits vert clair sont les endroits où nous effectuons un check

Il y a cependant ici quelques problèmes, qui ont à voir avec la taille du pas, si elle est trop courte, nous vérifions le même bloc plusieurs fois, et si elle est trop grande, certains blocs ne sont pas vérifiés, comme celui en bleu ci-dessus. Pour rémédier à ce problème nous pouvons user du fait que nous sommes dans un espace où toutes les zones sont espacées de façon équivalente. Il ne nous suffirait donc que d'avancer jusqu'à arriver à une bordure de bloc, et refaire de même pour la prochaine bordure, ce qui rend

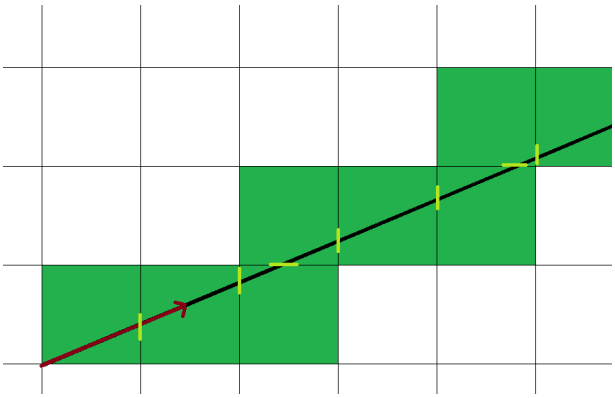


Figure 34: Schéma de la deuxième méthode envisagée pour regarder un bloc, avec la demidroite noire représentant la continuation du vecteur direction(rouge foncé). Les traits vert clair sont les endroits où nous effectuons un check

De cette méthode nous n'avons pas à craindre de faire trop ou pas assez de checks. La régularité ici des blocs est utile donc pour nous permettre de facilement prédire lorsqu'un changement d'un bloc à un autre se fera. Comment fait-on cela? En paramétrisant la demidroite noire avec le paramètre t, nous avons

$$\text{demidroite} = t \times \text{direction}$$

Et alors, en observant qu'après avoir traversé une bordure, la prochaine est croisée au bout d'un certain t, puis à nouveau au bout du même t en plus et ainsi de suite. Soit, à $nt_{x/y}, n \in \mathbb{N}$ une bordure est croisée en x/y avec $t_{x/y}$ représentant le t nécéssaire entre chaque passage de bordure de bloc dans une direction. Ceci ne marche cependant pas lorsque le joueur n'est pas placé à l'origine, dans quel cas $nt_{x/y}, n \in \mathbb{N}$ devient $nt_{x/y} - \text{pos}_{x/y}, n \in \mathbb{N}$. Nous avons donc maintenant un moyen de savoir quand la prochaine bordure est franchie, et il ne nous manque que l'équivalence direction-$t_{x/y}$. Celle-ci est relativement facile à obtenir.

Nous savons qu'à $t = 1$ nous avons le vecteur direction inchangé. Ainsi, le temps de franchissement dans une direction est de tps franchissement$_{x/y} = \frac{1}{t_{x/y}}$ (intuitivement, si le composant x de la direction est de disons 0.5, il faudra 2t pour arriver à x=1, puisque t=1 correspond à la direction inchangée, soit tps franchissement$_x = \frac{1}{0.5}$).
Il y a d'autres détails, mais ceci englobe les idées principales desquelles nous avons pu tirer notre algorithme, et donc notre implémentation en code

5.4.4 classe: Camera(GameObject)

Nous allons maintenant voir la classe caméra, qui va de pair avec le joueur (5.4.3), et qui est initialisée par world.world_manager.World (à 5.3.1. Elle se trouve dans le fichier world.player
Le rôle de la caméra est surtout de contenir toutes les informations nécéssaires au rendering.

def __init__ (self , position , direction , fov , aspect_ratio , clip=(0.1, 100), following_position=None,
Cette fonction initialise la caméra. La plupart des arguments ci-dessus sont pour l'initialisation du super(), le game_object.GameObject (à 5.4.2). Des variables en relation avec les paramètres additionnels sont donc initialisées. Voici donc les paramètres

param: fov
Ceci est le paramètre représentant l'angle de largeur du champ de vision de la caméra. Ci cela n'était pas clair, voici un schéma.

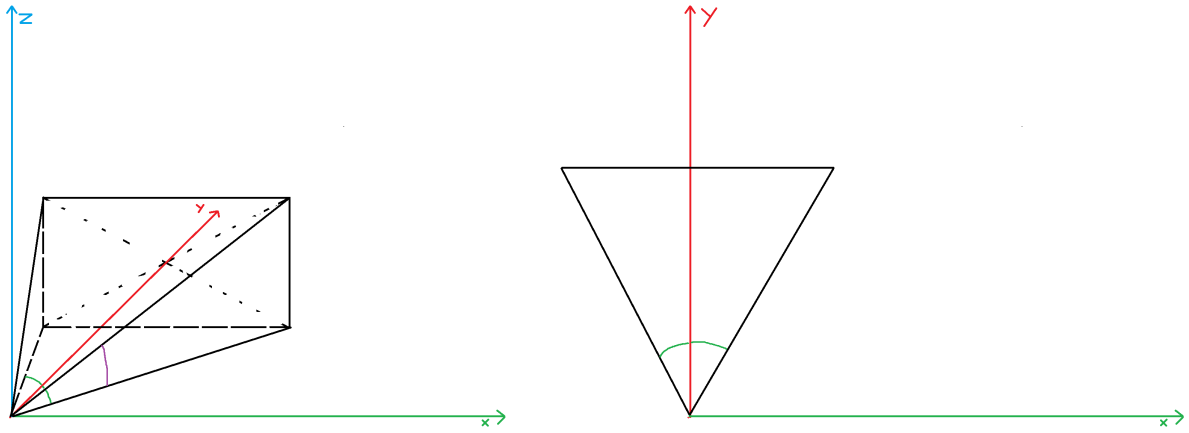


Figure 35: Un schéma de ce que le fov représente (l'angle en vert), avec le rectangle simulant un écran sur quoi les choses vont se projeter (ce n'est pas véritablement cela, voir la matrice de projection à 5.4.4

Le fov est donc à l'initialisation la mesure en degrés de l'angle en vert (nous la convertissons en radians plus dès lors mais les degrés sont plus visuels pour la plupart des gens). Vous vous demandez pourquoi peut-etre nous ne donnons pas l'angle rouge, et bien c'est parce que il dépend du fov ainsi que des dimensions de la fenêtre, le aspect ratio.
Valeur par défaut: 70

param: aspect ratio
Ceci est le ratio de $\frac{\text{window}_x}{\text{window}_y}$, qui est utilisé pour le fov (voir au dessus), et donc la matrice de projection (voir 5.4.4). Nous ne stockons cependant pas le aspect ratio tel quel, puisque nous utiliserons plus son inverse, et donc nous le stockons ainsi dans self.inverse_aspect_ratio. Défaut à 1.2

param: clip
Tout objet contenu entre clip[0] et clip[1] sera visible (voir la matrice de projection pour plus de détails à 5.4.4). Défaut à: (0.1, 100)

@property: def fov(self):
Nous faisons ici une propriété pour le fov (voir 35 pour voir ce que c'est), afin que nous sachions si il faut recalculer la matrice de projection ou non lors du prochain appel (si le fov a changé, la matrice n'est plus la même)

@property: def inverse_aspect_ratio (self):
Le aspect ratio = $\frac{\text{window}_x}{\text{window}_y}$ et donc inverse aspect ratio = $\frac{\text{window}_y}{\text{window}_x}$ Nous faisons ici une propriété pour le inverse aspect ratio (et non le aspect ratio car nous n'utilisons que 1/aspect_ratio), puisque si cette variable venait à être changée, il faudrait recalculer la matrice de projection, c'est donc ce que nous faisons, on fait en sorte qu'au prochain appel de la matrice de projection, elle soit recalculée.

@property: def projectionMatrix(self): et def _computeProjMatrix(self):
La propriété ne possède qu'un getter, puisqu'elle ne doit pas être modifiée. Lors d'un appel, si il y a besoin de recalculer la matrice de projection, la deuxième fonction est appellée. La dérivation de la matrice de projection est assez longue et assez complexe, donc voici où une dérivation peut être trouvée. Il y a cependant quelques changements par rapport à ce lien, notamment le repère. Pour passer de leur matrice à la notre nous avons donc fait (leur repère au notre):
$x \to x$
$y \to z$
$z \to -y$
En utilisant la notation que n=self.clip[0](near plane), f = self.clip[1](far plane), w=largeur de l'écran proche (r dans le lien), h = hauteur de l'écran proche(t dans le lien) Et donc la matrice de projection part de

$$\begin{bmatrix} \frac{n}{w} & 0 & 0 & 0 \\ 0 & \frac{n}{h} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Et arrive à

$$\begin{bmatrix} \frac{n}{w} & 0 & 0 & 0 \\ 0 & 0 & \frac{n}{h} & 0 \\ 0 & \frac{f+n}{f-n} & 0 & \frac{-2fn}{f-n} \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

@property def alpha_matrix(self): et @property def beta_matrix(self):
Lorsque nous projetons dans les shaders, nous utilisons la matrice. Mais cette dernière suppose le fait

que notre direction soit fixe, dans les +y, ce qu'elle n'est pas. En autres mots, quand nous avons la tête tournée, il faut tourner les points. Par combien?
Tournez votre tête vers la droite, c'est comme si de votre point de vue, votre direction était fixe, mais que les objets tournaient vers la gauche. Donc, il nous faut tourner les points par l'opposé de notre propre rotation. Combien vaut celle-ci? Et bien nous sommes tournés de α (voir à 24), ce qui fait que nous devons tourner les points de $-\alpha$. Nous avons alors nous même dérivé les matrices de rotation, mais vous les trouverez tout aussi aisément en ligne. Voici donc la matrice de rotation pour l'angle alpha:

$$\begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Ceci n'est cependant pas toute l'histoire, car nous avons certes résolu le problème de rotation "horizontale", mais il faut aussi tourner les points verticalement de $-\beta$, pour les mêmes raisons qu'avec α. Voici la matrice de rotation selon $-\beta$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) \\ 0 & \sin(\beta) & \cos(\beta) \end{bmatrix}$$

5.4.5 classe: game_chunk.Chunk

Cette classe se trouve dans le fichier world.game_chunk.py. C'est elle qui va être manipulée par le world_manager (à 5.3.1), et qui va afficher les blocs via les outils du rendering (4). Il est conseillé d'avoir lu la section sur l'affichage avant de lire cette partie

def __init__ (self , position , block_array):
Fonction appellée lors de l'initialisation d'un Chunk

param: position
La position de la chunk en coordonnées chunk/normalisées (voir 2.2.2)

param: block_array
Le np.array contenant les blocs renderables de la chunk (voir 2.2.3)

Lors de cette initialisation, la classe chunk va récupérer les blocs, et les mettre dans un ensemble, self.blocks (un ensemble puisque l'opération principale coûteuse effectuée est "x in self.blocks", qui est O(n) pour les listes et O(1) pour les ensembles, de plus, on évite de potentiels duplicats).
Ensuite, nous initialisons les variables:
self.cube_indices qui correspond à la liste des indices de chaque cube dans son chunkbatch.
self.cubes_dynamic qui correspond à la liste des tuples (x,y,z,id) de chaque bloc
Ces deux listes (cube_indices et cubes_dynamic vont de pair pour fonctionner comme un dictionnaire dans les deux sens, mais avec des fonctionnalités ajoutées
self.cubes_dynamic_set qui correspond à self.cubes_dynamic mais en ensemble, ce qui est plus rapide pour effectuer l'opération "x in a"
self.chunkbatches qui correspond au différents rendering.chunk_batch.ChunkBatch du Chunk
self.renderer qui correspond au renderer du monde **self.total_space** qui correspond au nombre de blocs stockables dans les différents chunkbatchs (on en crée un nouveau si nous nous en rapprochons)
self.indices correspond à l'ensemble des indices dans la scène des différents chunkbatchs du chunk (utilisé dans world.world._manager.World.unload_chunks à 5.3.1)

def load_to_screen(self, renderer):
Ceci va mettre les blocs contenus dans self.blocks dans un rendering.chunk_batch.ChunkBatch, et ainsi afficher les blocs. Les blocs ne seront toutefois pas ajoutés si leur id est corrompu, et auront alors un id au hasard parmi certains (utile pour repérer des bugs). On récupère finalement les indices de tous les cubes dans le chunkbatch, ainsi que l'indice du chunkbatch dans la scène.

param: renderer l'instance rendering.renderer.Renderer instanciée par le world_manager (à 5.3.1) dans laquelle le Chunk va mettre les blocs

def add_block(self, block):
Cette fonction permet de rajouter un bloc au chunk. Pour ce faire nous regardons si le dernier chunkbatch initialisé a encore de la place (plus précisément si il reste au dessus de 8 places théoriques, puisque nous avons eu des bugs où les places théoriques différaient très légèrement, entre 1 et 5 places en moins que prédites par le self.total_space (à 5.4.5)). Dans le cas affirmatif nous ajoutons le bloc traditionnellement, sinon, nous créons un nouveau chunkbatch, et nous rappelons Chunk.add_block(block) (et oui un peu de récursion ne fait pas de mal à personne)

def delete_block(self, block):
Cette fonction permet de supprimer le bloc désiré de la vram (pas du jeu, il faut modifier les variables chez le data manager et changer les fichiers en dur). Assez simplement, on récupère l'indice du bloc à supprimer dans son chunkbatch, et on le supprime de ce dernier. On met ensuite à jour quelques variables.

param: block
Le bloc à être supprimé, un tuple sous format (x_rel,y_rel,z_rel, id) avec les coordonnées relatives au chunk

5.5 La communication jeu-génération: les data_manager

Les data_manager sont une paire de programmes essentiels au bon fonctionnement du jeu: le data_manager_world et le data_manager_generation.
Leur existence est due à un problème assez simple, soit le fait que la génération d'un chunk prend du temps. Ainsi, si le jeu devait, à chaque fois que nous traversions une bordure de chunk, générer tous les chunks nouvellement visibles, attendre que cela se fasse, et puis les afficher, cela ferait des lag spikes de plusieurs secondes(5-10sec) à chaque traversée de bordure de chunk. Ceci n'est pas acceptable.
Ainsi, nous avons séparé les deux processus: le jeu, et la génération, en deux processus python différents, afin que la génération se fasse *en même temps* que le reste du jeu, ce qui permet de restreindre les lag spikes énormément.
Afin ensuite que les deux processus puissent communiquer, nous avons donc les deux data_manager, un du côté jeu (world) et l'autre côté génération, communiquant avec un multiprocessing.Pipe, sachant que le processus côté génération est initialisé par l'autre.
Mise à part cette communication, le data manager_world a d'autres fonctionnalités, telles que de stocker les données de chunks "actifs", soit les chunk très utilisées (la chunk du joueur par exemple)
La façon de laquelle les demandes de génération sont traitées est la suivante: une queue existe du côté de la génération, et le data manager generation essaiera alors de toujours générer la dernière chunk ajoutée (voir 5.5.2) Commençons donc avec

5.5.1 file: data_manager_world.py

Le data manager world est l'un des deux data manager. C'est celui qui sera (surprise!) du côté du jeu. Il sera toujours appelé par une variété de fonctions, dans plusieurs buts:
-Demander au data_manager_generation de générer du terrain.
-Récupérer les arrays de certaines chunks, et les stocker afin de ne pas avoir à réouvrir un fichier à chaque fois que nous avons besoin d'une chunk.
-Changer les arrays d'une chunk stockée dans le data_manager pour prendre en compte un évènement tel que le placement ou le cassage d'un bloc
-Sauvegarder une chunk stockée dans le data_manager en fichier, afin d'inscrire le changement pour de bon, et pour que ce dernier persiste après un reload de monde.
Ci-dessous est un schéma, qui représente l'échange entre les deux data managers via la Pipe, ainsi que les fonction handle_messages(), generation() et read_messages(), sachant qu'il est conseillé de lire d'abord en tout cas quelques fonctions clés dans les deux data managers auparavant, et d'avoir une bonne idée de comment le world manager fonctionne. Voici donc le schéma (vous pouvez zoomer)

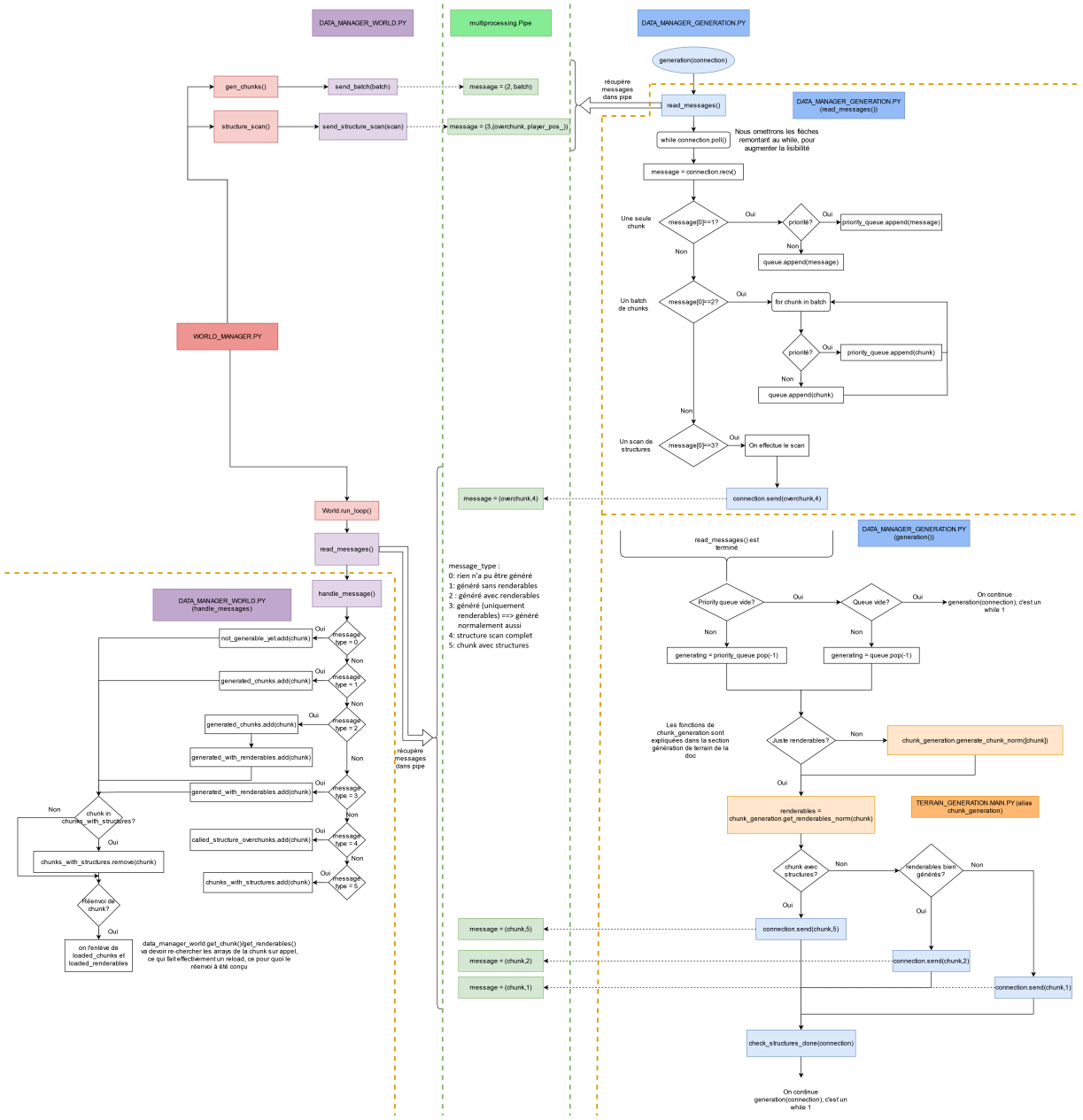


Figure 36: Schéma de la communication entre data_manager_world et data_manager_generation

Dans le schéma ci-dessus, nous avons une couleur par partie de programme ou programme, et les fonctions de ce programme colorées en cette même couleur, mais plus clair. Le world manager est la fonction qui appelle le data manager world, et il est lui même appellé par main (voir 5.2).
Nous avons ici également la section multiprocessing.Pipe au centre en vert, qui représente les messages envoyés d'un data manager à l'autre. Ainsi, les messages qui y sont envoyés le sont avec une flèche pointillée, et nous avons schématisé la récupération des messages avec une grande double flèche vers le Pipe.

Initialisation du fichier
Nous commençons par définir les variables o(ut)o(f)b(ounds)_volcanos_chunk_path et oob_chunk_path, en fonction du settings.WORLD_ID, qui sont les chemins vers les fichiers, qui, si ils existent pour une chunk donnée, veulent dire que la structure en question est présente sur la chunk.
Puis sont définies les variables
-generated_chunks = set() Ceci est l'ensemble de toutes les positions des chunks ayant été générées.
-generated_with_renderables = set() Ceci est l'ensemble de toutes les positions des chunks desquelles les renderables ont été générés.
Ensuite, nous itérons sur tous les fichiers contenus dans le fichier du jeu, et récupérons de là les chunks déjà générées (mais qui n'ont pas de structure manquante, voir has_structures à 5.5.1), et les ajoutons aux generated_chunks.
Nous faisons ensuite de même pour les renderables.
Puis sont définies encore d'autres variables:
-loaded_chunks = [ ] Ceci sont les positions des chunks affichés à l'écran (une liste puisque nous voulons éviter que la liste ne s'étale trop en mémoire, et donc devons enlever les éléments les plus ancients)
-loaded_renderables = [ ] Même chose que loaded_chunks, mais pour les renderables des chunks.
-loaded_chunks_dict = {} Ceci est un dictionnaire de paires (chunk_pos, np.array), il va de pair avec loaded_chunks
-loaded_renderables_dict = {} Ceci est la même chose que loaded_chunks_dict, mais pour les renderables
-called_structure_overchunks = set() Ceci est l'ensemble des overchunks ayant déjà été demandés (2.2.4)
-chunks_with_structures = set() Ceci est l'ensemble des chunks ayant des structures. Leur comportement peut différer de celui des chunks standard.
-changed_chunks = set() Ceci est l'ensemble des chunks à avoir subi une modification (cassage, placement), et qui doivent donc être sauvegardés en dur lors de leur désaffichage ou de la fin du monde.
-not_generable_yet = set() Ceci est l'ensemble des chunks considérés comme n'étant pas encore générables (si on nous dit qu'un tel chunk n'est pas générable à un moment donné il serait bête de redemander précisément celui-là) -last_sent = [] Ceci est une liste des chunks dernièrement envoyées, afin d'éviter de renvoyer tout le temps les mêmes chunk alors qu'elles ne sont pas générables (même principe que pour not_generable_yet mais plus général).

La dernière partie de la phase de l'initialisation de ce data manager est le lancement du processus de génération. Ceci ce fait donc en créant une multiprocessing.Pipe, soit un *canal* à travers lequel deux processus différents peuvent s'envoyer des messages.
Puis nous créons le processus avec multiprocessing.Process de la fonction data_manager_generation.generation (à 5.5.2), et le démarrons.

def has_structures(chunk):
Cette fonction détermine si la chunk demandée a une structure non générée sur elle (volcan ou village). Elle est utilisée dans la phase d'initialisation du data_manager_world. La fonction renvoie True si oui, et False sinon.

param: chunk
La chunk pour laquelle vérifier, en format (chunk_x, chunk_y), coordonnées chunk normalisées.

def send_to_queue(chunk, message_type=0):
Cette fonction va envoyer une chunk à la queue du côté génération des data manager (voir 5.5.2)
Cependant il faut faire attention si l'on veut demander plusieurs chunks, puisque ouvrir le Pipe pour envoyer un message, puis la rouvrir pour un autre etc. est beaucoup plus lent que d'ouvrir la Pipe une fois et tout envoyer d'un coup.
Si on cherche donc à demander à générer beaucoup de chunks à la fois, il vaudrait mieux appeler la fonction data_manager_world.send_batch

param: chunk
Ceci est la chunk à envoyer, un tuple (chunk_x, chunk_y) de coordonnées chunks normalisées

message_type Ceci est le type de génération demandé. Afin de minimiser les choses transmises à travers la Pipe, nous avons donc un int pour toutes les générations possibles, voici les identifiants:
0: génération normale
1: juste les renderables
2: génération normale, mais prioritaire (voir 5.5.2) 3: juste les renderables mais prioritaire
Nous n'envoyons toutefois pas (chunk, message_type) comme message, mais (1, chunk, message_type) puisque le message[0] permet au data manager generation de savoir s'il s'agit d'un single chunk send, d'un batch chunk send ou d'un structure scan

def send_batch(batch):
Cette fonction va envoyer plusieurs chunk à être générés toutes en même temps, ce qui est plus rapide que chunk par chunk. Le message alors envoyé sera de (2(identifiant de batch send), batch) avec batch un tuple de tuples : ((chunk_x, chunk_y), generation type)

param: batch Le batch de chunks à envoyer, sous format (((chunk_x, chunk_y), generation type),((chunk_x, chunk_y), generation type) ...)

def send_structure_scan(chunk_scan, player_pos):
Demande au data manager generation d'effectuer un scan de structures sur une grande aire (2.2.4). Si le chunk_scan n'avait pas déjà été effectué, nous envoyons alors au data manager generation d'envoyer (3(ceci est l'identifiant d'un structure scan), (chunk_scan, player_pos))

param: chunk_scan
Le structure scan à envoyer, voir 2.2.4

param: player_pos
La position du joueur, en coordonnées normales.

def read_messages(): Cette fonction va, tant qu'il y a des messages non lus dans ce côté-ci de la Pipe (connection.poll()), récupérer les messages (connection.recv()) et traiter le message (data_manager_world.handle_message juste en dessous). Un schéma est à 36
Attention, il est conseillé d'appeler cette fonction au maximum une fois par frame puisque la fonction connection.poll() est très coûteuse en termes de performance.

def handle_message(message): Cette fonction va prendre les messages reçus depuis le data manager generation via la Pipe, et en fonction du message de confirmation va mettre à jour différentes variables. Un schéma est à 36

param: message Le message reçu depuis le data manager generation, de format ((chunk_x, chunk_y), generation_id) ou (overchunk, generation_id) pour le cas d'un structure scan

le traitement du message Il y a 6 différents generation_id:
-0: rien n'a pu être généré, on ajoute la chunk à not_generable_yet
-1: généré sans renderables, on ajoute à generated_chunks
-2: généré avec renderables, on ajoute à generated_chunks, et à generated_with_renderables
-3: généré (uniquement renderables). Ceci implique une génération normale, donc sensiblement pareil que pour generation_id = 2
-4: structure scan complet, on ajoute à called_structure_overchunks
-5: chunk avec structures (à ne pas renvoyer, qui sera automatiquement renvoyé par le data manager generation tout seul - en théorie), on ajoute à chunks_with_structures
Pour les variables mises à jour, regardez 5.5.1. Il y a également un schéma à 36
Pour les cas 1, 2 et 3, nous regardons si la chunk était une chunk avec structures, dans quel cas on l'enlève de l'ensemble. Nous regardons également si la chunk était un réenvoi, dans quel cas nous supprimons les références à la chunk dans les variables, qui fera en sorte que la chunk soit re-cherchée lors d'un get_chunk ou get_renderables (2e et 3e fonctions en dessous)

def renderability(chunk):
Ceci est une fonction qui va déterminer si la chunk (de format (chunk_x, chunk_y) en coordonnées de chunk normalisées) peut voir ses renderables se faire demander (il faut que les 4 chunks voisinantes aient leurs chunks de générées, et que la chunk en question ait la sienne de générée également)

def get_renderables(chunk, generate, read):
Ceci est la fonction qui permet de récupérer le np.array associé aux renderables d'une chunk, et qui permet

de ne pas avoir à réouvrir le fichier à chaque fois que nous en avons besoin, puisque le data manager va stocker les arrays dans une liste (à longueur régulée).

param: chunk
La chunk de laquelle on veut avoir les renderables, sous format (chunk_x, chunk_y) en coordonnées chunk normalisées.

param: generate
Si le data manager doit demander à générer les renderables de la chunk dans le cas que ceux-ci n'existent pas. Ils seront alors demandés avec top priorité.

param: read
Si la fonction doit lire les nouveaux messages venus du data manager generation avant de s'éxécuter (ATTENTION, cela peut être très coûteux (5.5.1)

def get_chunk(chunk, read):
Ceci est la fonction qui permet de récupérer le np.array associé à une chunk, et qui permet de ne pas avoir à réouvrir le fichier à chaque fois que nous en avons besoin, puisque le data manager va stocker les arrays dans une liste (à longueur régulée).

param: chunk
La chunk dont nous voulons le np.array, sous format (chunk_x, chunk_y) en coordonnées chunk normalisées.

param: read
Si la fonction doit lire les nouveaux messages venus du data manager generation avant de s'éxécuter (ATTENTION, cela peut être très coûteux (5.5.1)

def change_in_renderables(chunk_x, chunk_y, new_renderables=None, block=None):
Cette fonction est appelée lors d'un changement dans une des chunks (placement/cassage). Elle va alors mettre à jour les renderables pour s'adapter à ce changement.
Cette fonction a deux modes d'utilisation: soit on donne un nouvel np.array (fait par fast_renderables) soit on donne un bloc à ajouter (fait par world_manager.World.place_block à 5.3.1)
Dans les deux cas, la chunk est ajoutée à changed_chunks (voir à 5.5.1)

param: chunk_x, chunk_y
Les coordonnées normalisées de la chunk qui nous intéresse

param: new_renderables
Si la 1ère option de changement est choisie, passer un np.array comme argument ici (les nouveaux renderables), sinon, laisser à None

param: block Si la 2ème option de changement est choisie, passer un tuple (x,y,z,id) qui sera alors ajouté aux renderables.

def change_in_chunks(chunk_x, chunk_y, bloc_pos, bloc_value):
Cette fonction est similaire à change_in_renderables (5.5.1, juste qu'ici l'option de passer un nouvel array est enlevée. Sinon, le fonctionnement est le même.

param: chunk_x, chunk_y
Les coordonnées normalisées de la chunk qui nous intéresse

param: bloc_pos, bloc_value
La position du bloc à changer, relative à son chunk, et d'identifiant bloc_value

def save_chunk(chunk):
Cette fonction est appellée lorsqu'un chunk est désaffiché entre autres, et elle va, si la chunk est une chunk qui a été changée (is in changed_chunks?), prendre la version des renderables et du array de la chunk et overwrite le fichier en dur

5.5.2 file: data_manager_generation

Nous avons maintenant vu le data manager world, et il nous reste donc le data manager generation. Ceci est moins un data manager qu'une porte de communication entre le data manager world et la génération, mais afin d'illustrer la dualité des deux programmes, nous avons décidé de garder cette nomenclature.
Il n'y a pas beaucoup de fonctions définies ici, mais certaines sont assez complexes, et sont donc schématisées à 36.
Le concept important de ce fichier est celui de *queue*. Essentiellement, c'est une liste dans laquelle le programme va récupérer des demandes de génération, envoyées depuis le data manager world, et de laquelle le dernier élément sera toujours généré en premier. Ceci permet donc d'avoir un ordre de priorité dans la génération. Il y a également une queue en plus, la priority_queue, soit une queue mais avec des éléments qui seront toujours générés avant ceux de la queue. Ceci est utile pour disons une chunk non générée mais qui devrait être visible: nous pouvons lui demander d'être générée en priorité.

def read_messages(connection):
Cette fonction va lire les messages contenus dans notre côté de la Pipe, et est appellée par data_manager_generation.generati
en dessous). Cette fonction va ensuite prendre chaque message et le stocker dans la priority queue, ou dans la queue normale, ou encore effectuer un chunk scan (en haut à droite dans le schéma 36)

param: connection
La connection avec le data manager world, donné lors de l'initialisation du Pipe

def generation(connection):
Cette fonction est la boucle de la génération, qui va demander à lire les messages du Pipe, qui va gérer les commandes dans les queues, et qui va renvoyer un messages de confirmation au data manager world. Il y a un schéma représentant le fonctionnement de cette fonction avec le data manager world à 36

param: connection
La connection avec le data manager world, donné lors de l'initialisation du Pipe

def check_structures_done(connection):
Cette fonction est appellée par data_manager_generation.generation à la fin de chaque itération. Elle va itérer sur toutes les chunks avec des structures n'ayant pas encore vu leurs renderables être générés, et va essayer de les générer (nous récupérons ces chunks en ouvrant un fichier json contenant précisément cette information.) Puis, en fonction du résultat de la génération des renderables, nous renvoyons ou non un message de confirmation au data manager world pour l'informer du fait que telle chunk avec structures est maintenant générée avec renderables, et que nous pouvons nous en servir. C'est en tout cas ainsi dans la théorie, mais la pratique s'est avérée être un peu plus capricieuse, et donc nous nous retrouvons à devoir redemander (côté world) des chunks avec structures

param: connection
La connection avec le data manager world, donné lors de l'initialisation du Pipe

def has_structures(chunk, structures_to_check):
Cette fonction va retourner un bool pour savoir si une certaine chunk a une structure non générée sur elle (True: il y a une structure). Cette fonction est utilisée dans check_structures_done() et dans generation(). La façon de laquelle nous vérifions ce fait est en regardant si un certain fichier lié à une structure non générée existe ou non. Le path de ce fichier est préétabli en début de file, avec les variables VOLCANO_LAVA_CHECK_PATH et VILLAGE_HOUSE_CHUNK_CHECK_PATH

param: chunk
La chunk pour laquelle on veut vérifier les structures présentes, en format (chunk_x, chunk_y) en coordonnées chunk normalisées.

param: structures_to_check
Un tuple de strs représentant les différentes structures pour lesquelles vérifier (Défaut = ("volcanos", "villages")), sachant que le jeu n'a pour l'instant que les structures de volcan et de village