ISOCRAFT STORY

Rapport de Soutenance

BERTRAND Nicolas, CHANE Romain,
CLERAULT Raphaël, DALOZ Léo

14 juin 2024

Table des matières

1 Introduction

Cette section faisait originellement partie du cahier des charges. Elle apparaît ici afin de faciliter la compréhension du rapport.

1.1 Nature du projet

IsoCraft Story est le fruit d'une inspiration née de l'immensité créative de *Minecraft*, alliée à la volonté de transcender les frontières du jeu de construction. Ce projet représente une fusion audacieuse entre le sandbox classique et une vision novatrice de l'exploration. Plongeant les joueurs dans un univers alien fascinant, le jeu se distingue par sa perspective isométrique, promettant une expérience de jeu qui repousse les limites de l'ordinaire.

Le joueur incarne un intrépide explorateur de galaxies s'étant écrasé sur cette étrange planète. Il est désormais chargé de reconstruire son vaisseau, tout en tentant de percer les secrets de cet environnement extraterrestre, à travers une exploration attentive et l'utilisation des ressources disponibles.

Mais le joueur est aussi encouragé à rester sur cette planète, à construire ce qu'il veut, de la petite maison en terre sablonneuse à la civilisation avancée grâce aux ressources qu'il peut trouver et assembler pour progresser avec ses amis en multijoueurs.

Un concept incontournable du jeu est la formation de la planète en "couches" parallèles et placées les unes sur les autres, formées de cartes du jeu uniques à chaque partie et abritant une faune et flore inédites. Ces couches servent de progression au joueur, qui s'aventure toujours plus profond dans cet univers hostile, à la recherche de son histoire, de matériaux pour son vaisseau, et surtout de l'aventure.

Voici des captures d'écran des deux premières couches :

1ère couche - surface, désert aride

2ème couche - forêt luxuriante

1.2 Notre équipe

Nicolas

Je suis Nicolas BERTRAND, 18 ans, possédant plusieurs passions dans le domaine du sport avec le Basket-Ball, dans le domaine de l'art avec le dessin. Mais aussi dans le domaine de la logique avec les mathématiques, les échecs et notamment l'informatique. Les jeux vidéo me sont aussi un centre d'intérêt fort depuis mon plus jeune âge.

Mon premier contact avec l'informatique fut au collège avec des activités en mathématiques où l'on devait utiliser scratch pour dessiner des formes géométriques et résoudre des problèmes. Même si ces activités étaient amusantes, ce n'est qu'en première que mon regard se tourna de plus en plus vers l'informatique. En effet mes spécialités de physique et de Sciences de l'Ingénieur

m'ont fait découvrir des langages de programmations tels que Python, Arduino et HTML.

Cela fait aussi un peu plus d'un an que mes connaissances et compétences en dessin se sont grandement approfondies. L'étude des perspectives, des couleurs, des lumières, des volumes et de l'utilisation de l'espace m'a permis d'acquérir une meilleure compréhension de la construction cohérente de mondes et de personnages. De plus l'utilisation régulière d'une tablette graphique m'a rendu plus à l'aise dans la création sur support digital et m'a amené une maitrise intermédiaire du logiciel de dessin Krita.

Mon attrait pour la logique mathématique, l'informatique et le dessin me serviront tout le long du projet sur Unity pour pouvoir créer le plus librement possible avec des camarades qui ont tous plus d'énergie et de passion les uns que les autres.

Romain

Je suis Romain CHANE, âgé de 17 ans, passionné d'informatique depuis mon plus jeune âge. La firme Nintendo m'a introduit au monde du jeu vidéo, que je considère comme un art capable de susciter de profondes émotions chez les joueurs. Actuellement, je suis fervent admirateur de la série des Souls, qui offre une expérience de jeu captivante et mémorable.

Au collège, j'ai été initié à la programmation à l'aide de Scratch, un logiciel qui m'a permis de concevoir divers jeux. Parmi ceux-ci, se trouvent deux jeux de combat, respectivement inspirés de Super Smash Bros et de Street Fighter, ainsi qu'un jeu de course en pseudo 3D mettant en vedette le célèbre hérisson "Sanic". Aussi, j'ai acquis une petite maîtrise des langages Python, assembleur, HTML, CSS, Java ainsi que des requêtes en SQL sur des bases de données au cours de mes études au lycée.

Dernièrement, j'ai découvert RPG in a Box, un logiciel dédié à la création de jeux de rôle. Bien que mes projets sur cette plateforme n'aient jamais abouti pour cause de motivation et que la création de jeux avec cet outil soit extrêmement limitée, j'ai apprécié le fait de pouvoir façonner mon propre univers dans un ordinateur.

C'est pour toutes ces raisons que j'attends avec impatience de débuter mon nouveau projet sur Unity, qui m'offrira une plus grande liberté de création, qui me permettra de découvrir et de me servir d'un nouveau langage de programmation, qui me contraindra cette fois-ci à le finir puisqu'une deadline est imposée

et surtout, qui me permettra de travailler avec des gens qui sont tout aussi passionnés que moi.

Raphaël

Je suis Raphaël CLERAULT, 18 ans, passionné d'informatique depuis mon plus jeune âge. J'ai des connaissances dans de nombreux langages comme le C, C++, C#, python, java, javascript, html/CSS, scratch, CAML, assembleur-x86, assembleur AVR, et mon propre assembleur, et mon propre langage de programmation.

Mes premiers projets, bien que modestes, ont été fait sur Scratch, puis Python, puis Unity, puis UE4/UE5, puis sur mon propre moteur de rendu. En 3e (2019) j'ai codé un jeu entier pour mon collège, à la demande de mon école, pour rendre les cours de français plus interactifs. J'ai ensuite, entre autres, codé une IA pour résoudre des parties du jeu *Démineur*, disponible sur le Play store au nom de *Démineur solveur*. J'ai ensuite porté *DOOM II* sur ma propre console portable, que j'ai fabriquée de A à Z, puis codé un moteur de rendu 3D pour cette console, très modeste de puissance (4000x moins puissante qu'un ordinateur moderne).

Léo

Je suis Léo, 18 ans. J'ai une expérience relativement variée dans la créa-tion d'applications et de jeux divers, souvent avec un moteur de rendu bas-niveau par rapport à Unity, comme par exemple : PyOpenGL, PyGame, tkinter (Python), Canvas (JavaScript), SDL/SDL2 (C++). Faire un jeu avec Unity change complè-tement le champ des possibles pour moi, avec l'approche totalement différente du moteur, de son contenu tout prêt mais qui ne va jamais parfaitement, c'est pourquoi je me retrouve souvent à recoder des mécaniques essentielles, par exemple un ray tracer, ou un système de collisions avec des équations différentielles pour juste un peu plus de performances.

J'aime lier les mathématiques à l'informatique, je l'ai par exemple fait quand j'ai créé mon ray tracer et différents systèmes de collisions, utilisé la trigonométrie pour diverses animations, interpolations et mondes générés procéduralement, mais aussi pour cracker de multiples jeux grâce aux probabilités et au dénom-brement. J'ai aussi profité de cette symbiose quand j'ai représenté des fractales, créé des moteurs 3D ou encore 4D avec des calculs d'intersections de divers objets mathématiques.

J'ai développé mes connaissances en programmation en grande partie grâce à ma soif d'apprendre et à Python. L'utilisation d'un langage complètement nouveau pour moi est, je le crois, une excellente voie d'apprentissage. Par ailleurs, ayant plus ou moins de connaissances dans une quinzaine de langages de programmation allant de HTML/CSS à brainfuck, en passant par l'expérience acquise pendant la réalisation de trois ordinateurs 8-bits dans *Minecraft* et plusieurs langages d'assembleur, ainsi qu'un compilateur C-like, je pense être capable de faire évoluer le projet.

2 Fonctionnalités techniques

Cette partie se focalise sur l'état du projet, et des directions adressées des points de vue technique et artistique. Elle constitue la majeure partie du rapport et permet de comprendre précisément le fonctionnement interne du jeu ainsi que les mécaniques qui s'offrent aux joueurs.

2.1 Génération de la map

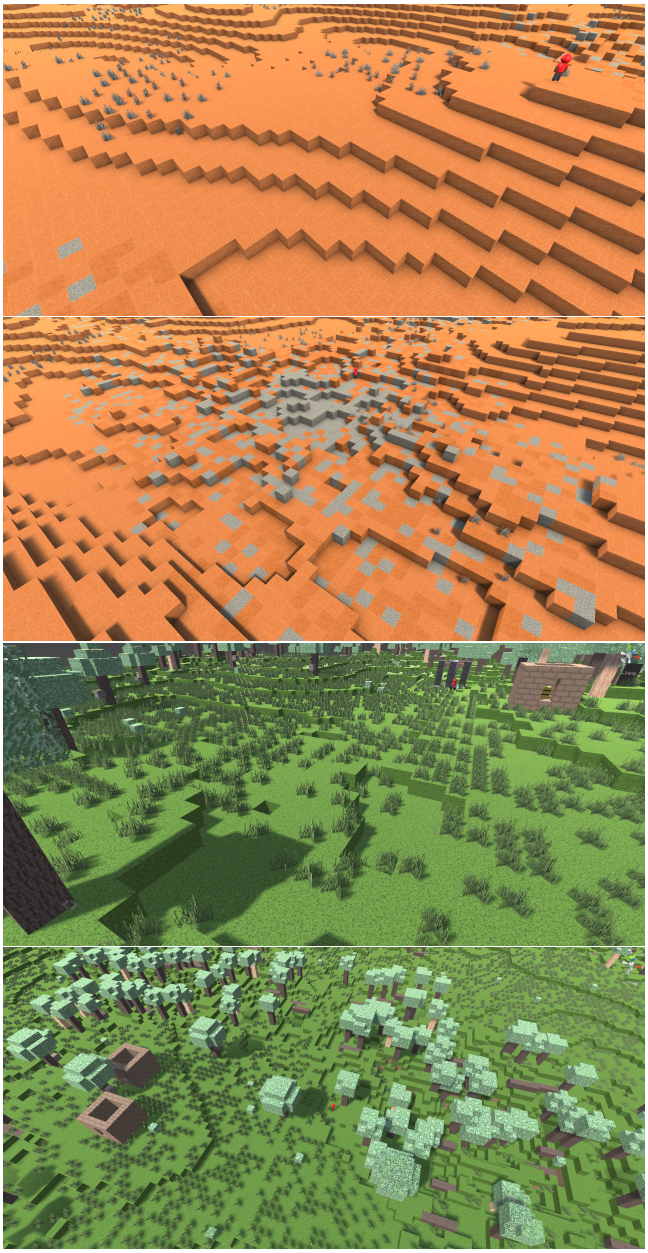
Le monde d'IsoCraft Story est constitué de différentes "couches", constituées de blocs, ou voxels. Ces couches sont de taille virtuellement infinie, d'une part, et il existe une grande quantité (plus de quatre milliards) de mondes uniques. Cette quantité doit ensuite être multipliée par le nombre de couches du jeu. Comment faisons-nous pour offrir cette expérience aux joueurs ?

2.1.1 Noise et génération

Comme expliqué dans le précédent rapport, nous utilisons une technique appelée "génération procédurale", qui consiste ici à ne pas stocker les différentes cartes du monde au préalable, mais à les "générer" en temps utile. Nous n'allons pas ici entrer plus en détail faute de place, mais nous dirons que nous utilisons la librairie *FastNoiseLite* pour obtenir une *heightmap*.

Une heightmap est ici un tableau bidimensionnel de valeurs entre -1 et 1, qui sont utilisées en tant que hauteur du terrain à chaque point vertical de la carte. Cela nous permet alors d'obtenir une variation rudimentaire de hauteur de terrain, qui est alors transformée selon nos besoins pour obtenir différents types de terrain. Ci-dessous quelques exemples : de haut en bas, un désert de

dunes, un champ de cratères de météorites, une plaine luxuriante et une petite forêt.



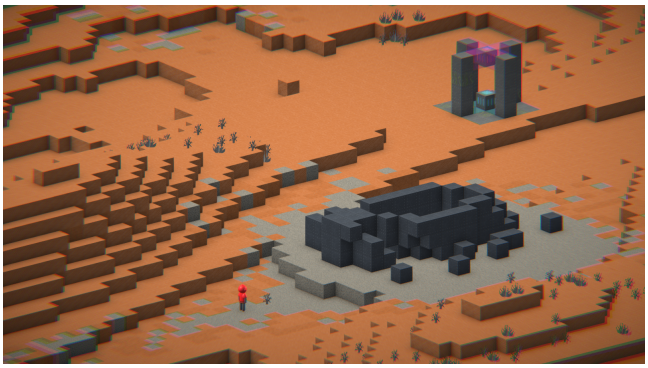
2.1.2 Structures

Le monde serait bien vide avec seulement ce processus de génération, c'est pourquoi des "structures", des assemblages de blocs prédéfinis, sont placés dans la carte. C'est le cas des arbres que vous pouvez voir sur les images ci-dessus.

Une certaine logique doit être utilisée afin de garantir par exemple qu'à la génération d'un chunk, des éventuels blocs "débordant" d'une structure située dans un autre chunk soient bien générés.

La plupart des structures sont placées aléatoirement selon des règles bien précises - par exemple, plus d'arbres dans les régions forestières, plus de buissons dans les régions plus plates - , mais certaines structures clés pour la progression de l'histoire sont placées à des endroits spécifiques.

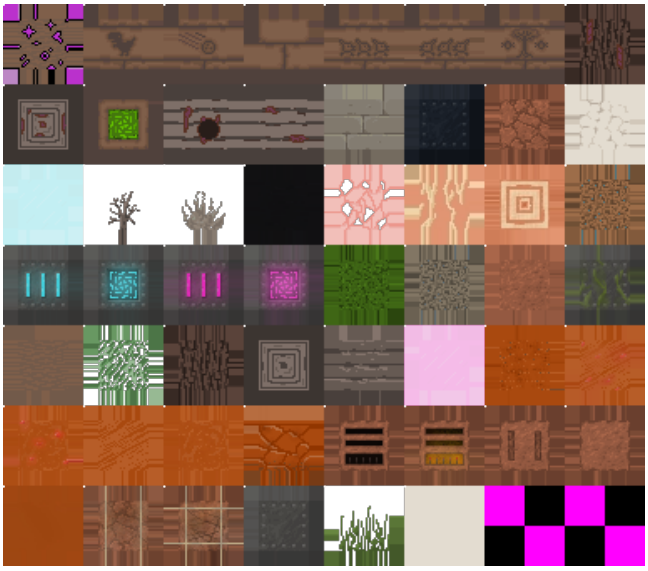
Par exemple, on peut voir ici le vaisseau du joueur écrasé. Le monde autour est aussi modifié : on peut voir que le terrain autour devient plat et que le choix de blocs change. Ceci est procédural, ce qui veut dire que par exemple le chemin de débris menant à l'épave sera unique à chaque partie, mais trouvera sa place dans le monde quelle que soit la génération.



2.1.3 Tex atlas

Comme vous pouvez le remarquer, les différentes couches sont composées de divers blocs. Ceci permet une grande créativité car le monde est facilement modifiable, comme nous le verrons plus tard. Mais ces blocs ont tous des textures différentes, souvent même plusieurs textures par bloc. Par exemple, une bûche a une texture pour les côtés, et une autre pour le haut et le bas.

A première vue, nous devons donc importer une grande quantité de textures et avoir une grande quantité de "meshes" - un mesh est un ensemble de points et de coordonnées dans une texture, constituant un object en 3D. Cependant, nous utilisons ce qu'on appelle un "tex atlas" ou une "texmap" : une grande texture comportant toutes les sous-textures que nous allons utiliser.



Lors de la création du mesh, on pourra venir sélectionner, pour une face donnée, les coordonnées dans le tex atlas correspondant à la texture voulue. Par exemple, disons que nous voulons rendre un bloc de briques, la texture correspondante se trouve à la position (0.52, 0.738) dans le tex atlas. Lors de l'affichage à l'écran, le moteur de rendu va chercher les pixels du tex atlas à cette position et va les rendre à l'endroit voulu dans la carte.

Voici les principaux avantages de cette méthode : d'abord, elle permet de ne charger qu'une seule texture dans la mémoire. Ceci limite les transferts de données entre le cpu et le gpu et accélère le rendu, augmentant le nombre d'images

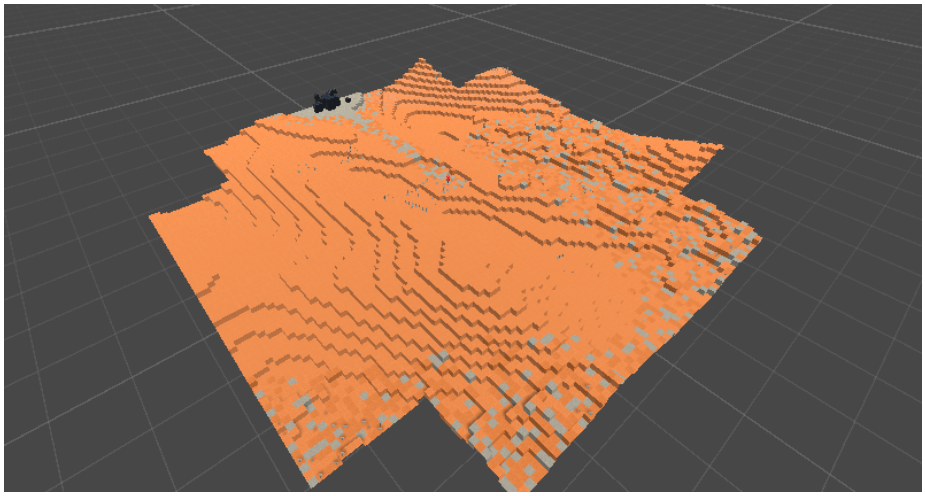
par seconde (fps) et donne un rendu plus fluide.

Ensuite, puisqu'il est simple pour l'ordinateur d'associer les textures des blocs au mesh de la carte, nous pouvons limiter grandement le nombre d'objets dans la scène. Cela améliore là aussi les performances, mais limite aussi l'occupation de la mémoire et limite le temps pris à générer une portion du monde.

2.1.4 Genération du mesh des chunks

Au final, nous pouvons donc afficher le monde d'une couche très facilement, et de manière optimisée. Cependant, comment faisons-nous pour afficher un monde de taille infinie ? Nous avons répondu à cette question dans le précédent rapport, mais en voici le point clé : le monde est divisé en "chunks" - portions - de 16x16x16 blocs du monde. Les chunks servent d'abord à la sauvegarde du monde, car ils nous permettent de sauvegarder seulement la portion du monde infini que le joueur a traversée et possiblement modifiée.

Mais ces chunks servent aussi à l'affichage du monde : à un moment donné, le joueur est positionné à un certain endroit sur la carte, à un certain niveau. Alors, les chunks qui vont être générés (ou récupérés d'un fichier de sauvegarde) et affichés pour ce joueur seront ceux dont la position est sufisamment proche du joueur. On peut voir ici un exemple de cette méthode. Bien que le monde soit infini, seulement quelques chunks sont générés à tout instant.



Penchons-nous plus en détail sur la génération du mesh de ces chunks. Là aussi, plusieurs optimisations sont possibles, afin de garantir la meilleure expérience utilisateur possible, d'ouvrir le jeu à une plus grande variété d'utilisateurs - ordinateurs moins récents, téléphones portables, consoles - et de pouvoir en retour fournir plus de contenu dans la plus grande marge de performances obtenue.

Nous devons afficher les blocs de la carte au joueur, et cela pour chaque chunk. Les blocs sont représentés sous forme d'un byte, leur identificateur unique (ID). Par exemple, un bloc d'air sera représenté dans la mémoire comme un 0, un tas de planches de chêne comme un 29. D'autres informations sont stockées, comme nous le verrons plus tard, mais pour le moment limitons-nous à cette information.

2.1.5 Tags

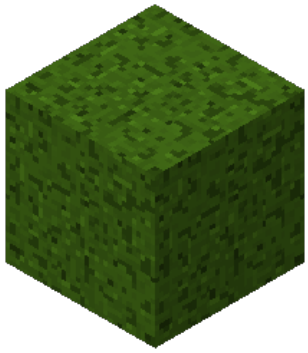
Lors de la génération d'un chunk, on pourrait naïvement afficher les 6 faces de tous les cubes qui ne sont pas de l'air, mais, comme expliqué lors du précédent rapport, des faces non visibles par le joueur seraient affichées, ce qui n'est pas optimal. Nous affichons donc seulement les faces des blocs en contact avec l'air. Mais nous nous sommes permis une plus grande liberté.

Chaque type de bloc a différentes propriétés, stockées une seule fois dans la mémoire pour s'y référer à tout moment. Nous appelons ces propriétés "tags". Ces tags nous permettent notamment de définir un comportement lors de la génération du mesh des chunks. Nous nous intéresserons ici en particulier aux tags NoTexture, Transparent, SemiTransparent, Is2D et IsModel :

— Le tag NoTexture est réservé aux types de blocs qui n'ont pas de texture. Par exemple, l'air a ce tag, mais ce n'est pas le seul, d'autres blocs - par exemple ceux de tag IsModel - n'ont pas de texture non plus.

— Le tag IsModel est réservé aux blocs dont le mesh n'est pas géré dans le chunk, mais autrement, car ils ne sont pas des cubes à 6 faces mais des modèles 3D conçus extérieurement. Ces derniers sont "posés" dans le monde en tant qu'objets à part entière, et n'obéissent pas aux règles classiques des blocs (tex atlas...).

— Le tag Is2D est pour les blocs qui ne sont pas des cubes, mais affichés sous forme de croix. Vous avez certainement remarqué ce comportement pour les hautes herbes dans les captures d'écran ci-avant, mais voici une comparaison entre un bloc de terre herbeuse et des hautes herbes (Is2D) :



Bloc de terre herbeuse

Hautes herbes

— Enfin, les tags Transparent et SemiTransparent sont pour les blocs dont la texture soit comporte des "trous" soit est partiellement transparente. C'est le cas par exemple des hautes herbes ou des feuillages, dont la texture n'est pas complètement "remplie", ou des blocs de verre à travers lesquels on peut voir.

Lors de la génération donc, les faces qui sont affichées sont celles des blocs opaques (qui ne sont pas Transparent), contigües par rapport à un bloc NoTexture. Cela permet de rendre possible une grande variété de blocs aux propriétés uniques, et d'accélérer le développement du contenu du jeu en n'ayant pas à constamment modifier la génération du monde pour répondre à nos besoins futurs.

2.1.6 Autres meshes

Les chunks ne comportent pas seulement un mesh, ils en ont en fait trois. Le premier est composé de la plupart des faces des blocs qui sont soit opaques soit avec certains pixels complètement transparents.

Le deuxième mesh est composé des faces des blocs semi-transparents. Cette décomposition en plusieurs meshes doit se faire de cette façon dû à la manière dont Unity gère la transparence. En effet, toutes les faces non transparentes sont affichées à l'écran en premier, puis toutes les faces semi-transparentes. Cela

permet à Unity de calculer la couleur des pixels de l'écran plus facilement, mais cela nous limite, car si nous placions toutes les faces dans le même mesh, certaines faces pourraient être dessinées devant d'autres, tout en étant plus loin de la caméra. Nous utilisons donc deux meshes différents.

Le troisième mesh est utilisé pour des calculs de collisions. Il est principalement utilisé lors de la modification du monde par le joueur : en effet, ce dernier clique sur un bloc pour le récupérer ou en placer un dans le monde, et ce mesh nous permet de savoir quel bloc a été cliqué pour obtenir cet effet de monde totalement destructible.

Là encore, cela nous offre une plus grande variété lors du développement du contenu du jeu, car nous pouvons avoir des blocs avec une texture, mais où le joueur peut passer au travers, ou des barrières invisibles au joueur.

2.1.7 Rotation des blocs

Nous avons évoqué le fait que les blocs ne sont pas seulement composés de leur ID, et qu'ils comportent d'autres informations. Une de ces informations est la rotation du bloc.

Afficher des blocs sous différents angles est assez simple, grâce à notre utilisation d'un tex atlas. En effet, nous donnons explicitement la liste des positions dans l'image à Unity lors de la génération du mesh d'un chunk, donc il suffit de changer leur ordre pour tourner une face d'un bloc.

Par exemple, imaginons que, pour afficher une certaine face d'un bloc, nous devons donner, dans l'ordre : bas gauche, bas droit, haut droit, haut gauche. Pour afficher l'image à l'envers, il suffit alors de donner à unity : haut droit, haut gauche, bas gauche, bas droit (soit décaler les positions de deux rangs).

Il devient plus difficile de savoir quelles faces doivent être tournées et de combien de degrés, ainsi que quelles faces vont devoir être déplacées à la place de certaines autres. En effet, si un bloc avec la texture A en haut est retourné dessus dessous, la texture A doit non seulement être retournée de 180°, mais elle doit aussi être affichée sur la face du dessous du bloc.

Pour cela, des tableaux de valeurs sont pré-générés afin d'être utilisés plus tard lors de la génération du mesh des chunks. Ces tableaux permettent d'éviter de calculer des opérations matricielles pour chacune des rotations, puisque les déplacements et rotations des faces sont les mêmes pour tous les blocs.

Grâce à ces différentes techniques et optimisations, le monde environnant les joueurs peut être affiché de manière fiable et rapide.

2.1.8 Modification du terrain

Le terrain est constitué de blocs pour être facilement modifiable par les joueurs. Comme dit précédemment, ils peuvent cliquer sur les blocs pour les détruire, ou en poser depuis leur inventaire.

Quand un joueur modifie le tableau de blocs dans un chunk, le mesh doit être recalculé. On relance alors simplement le processus de génération du mesh du chunk concerné, ainsi que celui des chunks voisins si le bloc avait été cassé sur le bord du chunk.

Ceci est l'une des raisons pourquoi tant d'optimisations étaient nécessaires, car un joueur peut modifier le contenu d'un chunk fréquemment lors du jeu, et qu'il est nécessaire de re-générer complètement le mesh du chunk concerné, n'étant pas possible de savoir simplement quel endroit du mesh a été modifié.

2.2 Sauvegarde

Nous avons évoqué précédemment la sauvegarde des chunks, mais nous n'avons pas précisé comment les différentes composantes des chunks sont stockées et récupérées des fichiers de sauvegarde.

2.2.1 Blocs et block entities

Les blocs sont généralement composés simplement d'un ID dans un tableau de nombres constituant un chunk. Mais certains sont plus complexes. Nous allons appeler ceux-ci des "block entities". A la création de ces blocs dans le mesh de leur chunk, des scripts vont aussi être lancés. Cela permettra d'obtenir des blocs avec lesquels des interactions sont possibles, ou du stockage d'inventaire. Par exemple, un coffre est un block entity, on peut cliquer dessus pour l'ouvrir et stocker des ressources.

Bien sûr, pour pouvoir sauvegarder les ressources placées dans ce coffre par exemple, ces ressources doivent être intégrées dans les fichiers de sauvegarde.

De plus, nous avons décidé d'utiliser le même format pour les blocs des chunks et le contenu des objets dans l'inventaire du joueur. Ces derniers sont aussi stockés sous forme d'ID, mais n'ont pas de rotation. Cependant, eux aussi peuvent lancer un script, par exemple si un objet est une carte du monde, cliquer dessus va ouvrir une carte, et si c'est un outil, détruire un bloc va user cet outil.

Toutes ces informations doivent donc être intégrées dans un seul et même format, qui doit être rapide à implémenter pour accélérer les phases de chargement. Le format suivant est donc adopté :

Pour les chunks, tous les blocs le constituant sont encodés puis concaténés pour former une seule chaîne de caractères. Les blocs sont encodés selon leur type :

— Si un bloc n'a pas de script, il est alors seulement composé d'un ID et d'une rotation. L'encodage consiste alors simplement à ajouter deux caractères, un pour chaque propriété. Cet encodage occupe donc deux bytes par bloc, ce qui est très peu et représente la majeure partie des blocs du jeu.

— Si un bloc a un script, ce script doit fournir une fonction d'encodage et de décodage. L'encodage du bloc va donc être constitué des deux propriétés du bloc - ID, rotation -, suivis de l'encodage du script.

— Si le bloc est un objet d'inventaire, le même format que précédemment est utilisé - ID, rotation, encodage du script - à ceci près qu'un caractère spécial est inséré entre la rotation et l'encodage du script, pour signaler lors du décodage qu'ici est stocké le script d'un objet, et non d'un bloc.

— Il est possible qu'un bloc aie à la fois un script d'objet et un script de bloc, dans ce cas le format prévoit aussi un encodage : ID, rotation, encodage du block, caractère spécial, encodage de l'objet.

Il est important de comprendre ce format, car il est essentiel à la fois pour comprendre le système de sauvegarde, ainsi que le partage en multijoueurs, comme nous le verrons plus tard. De plus, un encodage similaire est utilisé pour les entités présentes dans le jeu.

2.2.2 Sauvegarde du joueur

L'avantage de ce format est qu'il permet de sauvegarder à la fois des objets d'inventaire et des blocs de la carte du monde. Cette souplesse est même néces-

saire, car un bloc dans un chunk peut très bien être un block entity comportant un inventaire (exemple : un coffre), et l'encodage de ce coffre va nécessairement comporter des objets d'inventaire. Il faut donc que le format d'encodage supporte tous ces types de donnée.

La sauvegarde du joueur est alors assez simple, pour ce qui est de son inventaire : tout comme un chunk, son inventaire est encodé sous forme d'une chaîne de caractère formée de la concaténation de tous les blocs qu'il comprend.

D'autres informations sont stockées dans le fichier de sauvegarde du joueur, comme sa position, rotation, ses points de vie...

Bien sûr, le système de sauvegarde est très robuste, et il est probable que la majeure partie des modifications manuelles ou corruptions sera gérée efficacement : un exemple simple est la suppression d'une partie du fichier de sauvegarde : les données saines vont être chargées correctement, et les données corrompues vont être réinitialisées aux valeurs par défaut.

Cela implique que la génération d'une nouvelle partie est très simple : il suffit de générer une sauvegarde à partir d'une chaîne de caractères vide, ce qui va définir toutes les propriétés aux valeurs par défaut.

2.2.3 Tokens

Vous avez sans doute remarqué que le format de stockage d'un bloc dépend de s'il possède un script. Pour faire la différence entre ces cas, des "tokens", ici des caractères spéciaux, sont utilisés pour indiquer un certain format lors du décodage.

Ainsi, pour les blocs ou objets comportant des informations supplémentaires, leur encodage est entouré d'une paire de caractères spéciaux. Lors du décodage, on va donc regarder le premier caractère du bloc à décoder. Il nous indiquera quel format a été utilisé et la taille en caractères du bloc ou de l'objet.

Cela pose cependant un problème : disons qu'un des tokens est le caractère saut de ligne `\n`. Lors de l'encodage, il est possible qu'une valeur encodée comprenne ce même caractère `\n`. Pour éviter que ce caractère soit pris pour un token, il est remplacé par un substitut lors de l'encodage, puis est restauré lors du décodage. Cela nous permet de stocker toutes les informations nécessaires et de garder le format de tokens afin de décoder les informations correctement.

2.2.4 Régions

Accéder à des fichiers de sauvegarde pour chaque chunk n'est pas optimisé, aussi nous utilisons un concept que nous appelons "région". Une région est un assemblage de 8x8 chunks et est gardée dans la mémoire grâce à un système de cache de capacité bien définie. Dès lors, ce ne sont plus les chunks qui sont stockés dans les fichiers de sauvegarde, mais les régions.

De ce fait, lorsqu'un joueur a besoin de générer un chunk, il va chercher s'il est déjà mis en cache - si la région le comprenant est déjà dans la mémoire vive - , et dans ce cas la récupération du chunk est plus rapide car il n'y a pas besoin d'ouvrir un fichier. Si la région n'est pas chargée en mémoire, le fichier correspondant va alors être chargé dans le cache.

La génération des chunks est aussi modifiée : lors de la requête d'un chunk, si la région correspondante n'a pas encore été créée, les chunks qu'elle comprend vont tous être générés à ce moment-là. Ainsi, la génération se fait d'un coup et n'aura pas à être refaite pour tous les chunks de la région. Cela augmente les performances car, la plupart du temps, les chunks seront déjà générés, même si c'est la première fois qu'un joueur en fait la requête, et les seules grosses utilisations de ressources seront rares.

Un dernier point à soulever est que, à intervalles réguliers, le serveur sauvegarde l'intégralité du contenu du jeu pour limiter les pertes en cas de crash.

2.3 Multijoueurs

IsoCraft Story est un jeu qui peut et est encouragé à être joué en multijoueurs.

Chaque partie, solo ou multijoueurs, est en réalité une partie en multijoueurs. Une partie solo est simplement limitée à 1 joueur, et donc n'accepte pas d'autre connection. Dès lors, il est facile de passer d'une partie solo à une partie multijoueurs, simplement par la pression d'une case à cocher dans les réglages. Cela permet une expérience utilisateur plus simple d'une part, et permet de rendre le code plus compact, car toute partie est alors gérée comme une partie en multijoueurs. Une partie solo a donc un client et un serveur, tous deux sur la même machine.

La gestion du multijoueurs est centrée autour de la gestion des permissions pour limiter les risques de corruption par des utilisateurs malveillants ou mal-

encontreux, et de la mise à jour du monde et de son contenu pour tous les joueurs.

Cette gestion est comme suit : généralement, un client a un accès limité au contenu du monde : il n'a pas accès aux fichiers de sauvegarde, aux mobs, et ne peut donc pas risquer de corrompre la partie, le jeu des autres joueurs ou même le serveur. Un client gère les actions de son joueur, l'affichage du terrain, l'affichage des entités, et il n'a donc le contrôle que sur son joueur.

Le serveur, dans le cas contraire, gère la mise à jour ("ticking") de la carte. C'est lui qui va gérer le pathfinding des ennemis par exemple, et va récolter de tous ses clients toutes les informations sur tous les joueurs. Il va ensuite redistribuer ces informations en read-only aux clients.

De cette manière, la sécurité est garantie, contre les attaques ainsi que contre les problèmes internes au jeu, dont leur impact est limité. Cependant, cela signifie que nous devons gérer toutes ces interactions manuellement, car notre jeu fonctionne d'une manière unique et trop complexe pour utiliser les fonctionnalités toutes prêtes de Unity et Mirror - librairie multijoueurs pour Unity - tout en gardant un contrôle des informations, de leur sécurité et de l'optimisation des ressources.

2.3.1 Partage du terrain

La carte est stockée, pour des raisons de sécurité, seulement sur la machine hébergeant la partie, comme dit plus tôt. Voici comment se déroulent les requêtes pour afficher et interagir avec le monde :

Affichage du terrain

Lorsqu'un joueur se connecte, il aura besoin que le terrain autour de lui soit affiché. Il va donc faire une requête au serveur pour tous les chunks qu'il souhaite afficher - dans un certain rayon autour de lui. Le serveur va alors recevoir les différentes requêtes et effectuer le processus décrit plus tôt de récupérer ou générer la région contenant le chunk (encode) demandé, puis va renvoyer ces données au client.

Le client va alors décoder la chaîne de caractères récupérée, puis construire le mesh du chunk. Le client aura alors accès au monde autour de lui.

Modification du terrain

Lors de la modification du terrain - détruire un bloc, en placer un - par un client, le client va encoder le chunk modifié puis l'envoyer au serveur. Le serveur va alors modifier le chunk, soit dans sa région si elle est toujours chargée, soit dans le fichier de sauvegarde, pour garantir qu'il a la version la plus à jour du chunk. Le chunk, modifié et encodé, est ensuite renvoyé à tous les clients pour qu'ils aient eux aussi la version du chunk la plus récente.

Le fait que la demande passe par le serveur permet la gestion de plusieurs conflits d'accès : tout d'abord, puisque la commande est exécutée sur un seul thread (les requêtes sont exécutées une à une et non pas parallèlement), si deux clients placent un bloc au même endroit et au même moment, l'une des requêtes va être exécutée en premier. La deuxième, cependant, va échouer, car un bloc aura déjà été placé ici. Il n'y a pas de corruption du monde.

De plus, un autre conflit est adressé : imaginons qu'un client modifie son jeu pour lui permettre de détruire des blocs indestructibles. S'il essaye de détruire un bloc impossible à détruire, le fait que la modification du terrain passe par le serveur permet à ce dernier de rejeter la requête de modification du terrain.

Le bloc ne va donc pas être détruit, et la tentative de triche aura été adressée. Cependant, certaines actions pourront être effectuées sur la machine du client essayant de tricher, par exemple il pourra passer par une éventuelle ouverture effectuée par la destruction du bloc indestructible. Mais nous n'avons pas moyen de contrôler efficacement et de manière fiable la position locale des clients sans une latence variable, donc ce problème n'est pas adressé. Cela ne posera au moins pas un problème aussi conséquent que si le tricheur avait pu modifier la carte du jeu pour tous les joueurs.

Block entities

Les block entities, rappelons-le, sont des blocs qui ont la possibilité d'exécuter un script à leur position. Cela rend possible par exemple l'existence des coffres, blocs avec un inventaire. Dans ce cas, chaque client a un script exécuté pour chaque block entity, et lors de la modification de son contenu, le comportement est le même que lors de la modification du terrain : le chunk est encodé et envoyé au serveur pour informer tous les clients du changement.

2.3.2 Gestion des joueurs

Le monde étant séparé en couches, certains joueurs doivent être cachés, par exemple s'ils ne sont pas sur la même couche qu'un autre joueur. Ce phénomène appraît aussi si deux joueurs sont très éloignés l'un de l'autre : il n'y a pas besoin

de chercher à les faire s'afficher mutuellement si l'on sait qu'ils ne vont pas être visibles sur l'écran de l'autre joueur.

Nous ramenons sur le tapis les limitations qu'impliquent l'utilisation des manières de procéder simples de Mirror et Unity : ici, la solution la plus simple à utiliser est d'utiliser les comportements de base du moteur, qui consistent en la copie des joueurs chez tous les clients, puis à les cacher s'ils sont invisibles.

Dans ce cas, pour chaque joueur, un client aurait, chargé dans la scène, tous les objets de tous les joueurs ainsi que tous leurs composants (caméra, système d'écoute de son...) complètement inutiles au gamelpay, mais occupant de la mémoire. De plus, puisque les scripts des autres joueurs seraient tous actifs, il faudrait modifier le code pour ignorer les parties destinées seulement au personnage du client (gestion des touches et déplacement, gestion de la caméra...).

Pour éviter tous ces problèmes d'optimisation, les joueurs sont gérés autrement : un client va envoyer des informations le concernant (position, rotation, animation en cours...) au serveur. Le serveur aura donc une copie des informations essentielles à l'affichage de chaque client, qu'il va pouvoir renvoyer à tous les clients.

A ce moment, les clients pourront choisir efficacement quels objets afficher en fonction de s'ils leur sont visibles, et ne créer que des objets "vides" destinés seulement à avoir un retour visuel sur l'état des autres joueurs.

Ainsi, à tout moment dans la scène, une seule instance du joueur est active, et pour tous les autres joueurs un objet presque vide les représente.

Ce système facilite aussi la gestion des messages de connection/déconnection dans le chat, car on sait à tout moment si un joueur est présent : il a envoyé des informations au serveur dans un court espace de temps. On gère ainsi aussi de cette manière le timeout (un joueur a eu un problème de connection et il a été déconnecté du serveur automatiquement).

2.3.3 Gestion des entités

Les ennemis et autres mobs sont dirigés par un système de pathfinding et autres, comme nous le verrons dans le paragraphe concerné. Celui-ci est exécuté sur le serveur, pour éviter d'avoir à le calculer pour chaque client et d'avoir des désynchronisations.

Régulièrement, les informations nécessaires pour afficher les entités sont en-

voyées aux clients, d'une manière similaire à la gestion des joueurs. De plus, les clients envoient au serveur les requêtes concernant les interactions avec les entités (frapper, parler, échanger...).

Ces mobs doivent maintenant être incorporés dans le système de sauvegarde. Pour cela, le chunk dans lequel ils sont est déterminé, puis un système similaire au système de *garbage collector* est utilisé pour savoir quand les charger ou décharger.

A tout moment, le système de gestion des mobs sait si un client a besoin d'afficher un certain chunk, et donc certains mobs. Lors de la requête d'un chunk par un client, les mobs déjà dans la sauvegarde de ce chunk sont ajoutés si besoin, et lors de la mise à jour de tous les mobs, le serveur vérifie que tous les mobs sont à une distance raisonnable d'au moins un client. Sinon, cela veut dire que plus personne n'a besoin de les afficher, donc les entités concernées sont sauvegardées puis détruites.

De la même manière que pour les joueurs, un minimum d'opération est effectué et l'utilisation des ressources est limitée, tout en garantissant une mise à jour des entités fiable et égale pour tous les clients.

2.4 Mobs

2.4.1 Général

L'on appellera par la suite "mobs" tous les personnages du jeu, non-joueurs.

Comme expliqué brièvement plus haut, chaque mob possède un modèle 3D et un système de collisions et de déplacement en cohérence. La particularité des mobs est qu'ils peuvent se déplacer et attaquer. Ils ont donc les propriétés suivantes :

— Le nombre de point de vie
— Le nombre de dégâts par coup
— La vitesse d'attaque
— La distance d'attaque
— La force de saut

Les interactions entre les mobs consisteront donc à attaquer et être attaqué. Les mobs ont aussi une manière de se déplacer, la plupart, notamment le joueur, se déplaçant en marchant à hauteur constante et peuvent sauter lorsqu'il faut

franchir un obstacle.

Par exemple, les Bopako (à gauche) sont une sorte d'oiseau, donc ils peuvent sauter à hauteur variable en accord avec le fait qu'ils possèdent des ailes.

Les Oak Boka (à droite), quant à eux, ayant des jambes, se déplacent en sautant uniquement.

Bopako

Oak Boka

2.4.2 Idle state

Les mobs n'ont pas toujours une cible vers qui aller. Lorsque c'est le cas ils sont dans un état dit de repos (Idle state). Dans cet état, le mobs ne calculent pas de chemin menant à une cible. Le chemin qu'ils vont donc prendre dans cet état va être calculé de manière pseudo-aléatoire pour donner plus de vie aux mobs.

La génération du chemin diffère tout de même selon le mob. Ici, le chemin se répète tant que le mob n'a pas de cible.

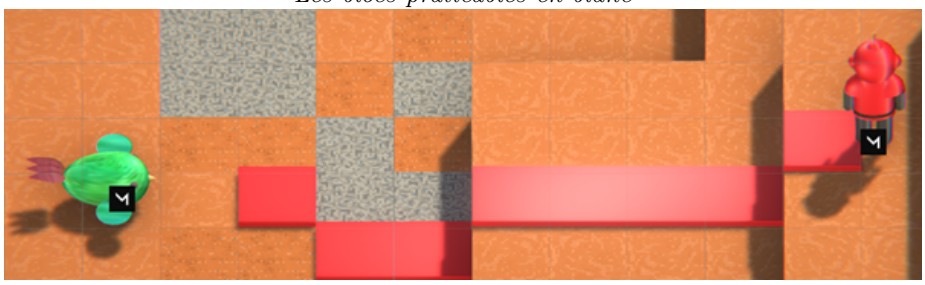
2.4.3 Pathfinding

Lorsque les mobs ont une cible à tracker, ils doivent pouvoir trouver un chemin menant à cet dite cible à l'aide d'un algorithme de pathfinding. Le monde d'IsoCraft Story étant composé de blocs, l'implémentation d'un pathfinding *A Star* est intéressant puisqu'il nécessite d'avoir un monde représentable sous forme de grille.

Voici son fonctionnement : On part de la position du mob, c'est à dire du bloc sur lequel il se trouve, et on analyse les 8 blocs autour de celui-ci (on trace un "anneau" autour do mob sur le sol). Pour chacun d'eux, s'ils sont praticables par la créature non-joueuse, ils seront stockés en tant que blocs praticables, avec une valeur associée qui correspond à la distance entre ce même bloc et la cible du mob, et généralement avec le bloc précédent.

On continue cette opération avec le bloc praticable ayant la plus petite distance à la cible jusqu'à tomber sur le bloc de la cible. Une fois cela fait, on peut définir le chemin praticable par le mob à l'aide de la chaîne de blocs précédents.

Les blocs praticables en blanc



Le chemin en rouge

Si aucun chemin n'est trouvé au bout d'un grand nombre d'itérations, on considère qu'il n'existe aucun chemin et le mob sera en état de repos (Idle state).

Une fois le chemin créé, il faut l'emprunter. Concrètement, un chemin est une pile de paires coordonnées/hauteur de saut - 0 le mob ne saute pas, 1 il saute d'un bloc, 2 il saute de deux blocs. Le mob va donc suivre successivement les coordonnées de la pile et sauter au bon moment et à la bonne hauteur pour arriver à la position de la cible.

Cependant, la position de la cible n'étant pas fixe, il est nécessaire de recalculer un nouveau chemin assez fréquemment pour que le mob n'aille pas à

une position de la cible trop ancienne. De plus, lorsque l'on recalcule un nouveau chemin, on supprime les anciens blocs praticables car le monde a pu être modifié, notamment par le joueur.

Chaque mob a des caractéristiques, comme la distance d'attaque ou la force de saut, qui vont permettre de déterminer si un chemin est praticable, résultant en un ensemble divers de mouvements pour les différents mobs de notre jeu, les rendant ainsi uniques.

D'autres algorithmes plus simples ne permettant pas toujours de trouver un chemin praticable sont implémentés pour créer des mobs avec des niveaux d'intelligence variés, variant ainsi l'expérience des joueurs.

2.4.4 Comportement des mobs en pratique

Les mobs étant des créatures hostiles dans IsoCraft Story, leur cible est par défaut le joueur le plus proche. Un mob peut tout de même changer de cible pour un autre qui l'attaque, ceci est utile pour l'ajout de nouveau mobs qui n'attaqueront pas uniquement le joueur.

Si la cible est présente dans le rayon de ciblage du mob, celui-ci se dirigera vers lui jusqu'à ce qu'il soit à sa portée d'attaque, sinon il sera en état de repos et n'exécutera aucune action particulière si ce n'est se déplacer de manière aléatoire.

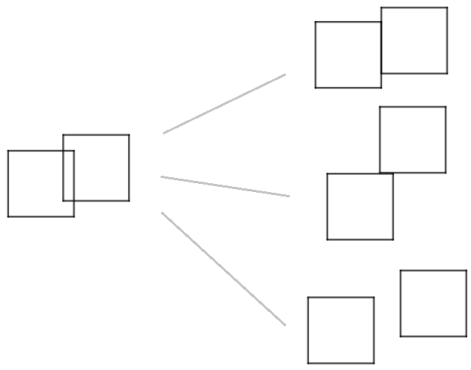
2.5 Déplacement du joueur

2.5.1 Système de collisions

Comme dit précédemment, notre jeu ou nos capacités en code ne nous permettent pas d'utiliser les fonctionnalités de base de Unity pour obtenir un fonctionnement optimal. C'est pourquoi nous avons recréé le système de déplacement et de gestion des collisions de Unity.

Rappelons-le, une "collision", dans le monde du jeu vidéo, se réfère à un moment où deux objets du monde occupent le même espace. Or, puisque la majorité des jeux aspirent à un certain réalisme ou du moins à une expérience utilisateur instinctive, certaines propriétés du monde réel sont recrées plus ou moins fidèlement. Dans le monde réel, deux objets ne peuvent occuper la même place. C'est pourquoi, lorsque cela arrive dans un jeu vidéo, l'on essaye souvent de trouver comment déplacer les objets en contact pour ne plus qu'ils occupent le même espace.

Voici un diagramme de deux carrés en collision à gauche, ainsi que différentes manières de régler le problème de collision à droite. Le but d'un système de collision est de trouver la meilleure solution à ce problème, ici la première.



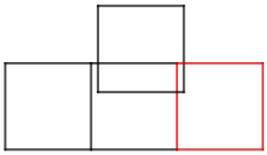
Ce problème est un problème largement adressé, car assez complexe. Le monde d'IsoCraft Story étant principalement constitué de blocs, les collisions sont relativement simples à résoudre par rapport à ce qu'effectue le système de base de Unity, c'est pourquoi nous avons aussi la possibilité de concevoir un algorithme plus efficace car plus adapté.

Nous sommes tout de même passés par plusieurs itérations du système de collisions avant d'en obtenir un satisfaisant, la tâche étant relativement complexe. Il existe peu d'informations satisfaisantes et faciles à trouver sur le sujet en ligne, donc nous utilisons une approche "maison" qui n'est pas parfaite, certains bugs apparaissent toujours même s'ils sont de plus en plus rares.

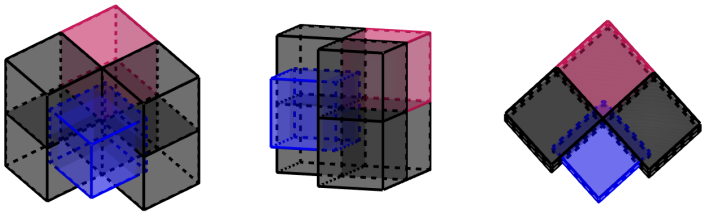
Nous ne décrirons pas ici en détail le fonctionnement interne de l'algorithme utilisé, mais voici tout de même quelques exemples afin d'illustrer certains problèmes rencontrés :

— Imaginons le cas vu ci-dessus, avec les deux carrés. Nous pouvons par exemple effectuer la correction gauche-droite, ce qui nous donnerait un résultat optimal, mais imaginons le cas à droite dans la figure ci-dessous, où il faut ici effectuer la collision haut-bas. Les deux cas sont presque identiques, mais la correction doit être identifiée différemment. Dans ce cas, la solution envisagée est de corriger dans la direction avec le moins de pénétration.

— Voici un autre exemple, où ici, en utilisant la méthode vue précédemment, les cubes du sol en bas et à gauche effectueront les corrections attendues, mais le cube rouge pousserait le joueur en haut, l'empêchant d'avancer. La solution utilisée ici est d'ignorer les faces situées entre deux blocs pour le calcul de correction.



— Ce dernier exemple illustre un problème venant de l'implémentation précédente : un monde en trois dimensions apporte des problèmes supplémentaires dans la résolution des collisions. Ici, le cube rouge ne peut effectuer de collision avec le joueur en bleu que par sa face supérieur, car les autres faces concernées sont en contact avec d'autres blocs, donc leurs corrections sont ignorées. La correction apportée par le cube rouge est ici de monter le joueur au niveau de sa face supérieure. Dans ce cas, le joueur est téléporté au-dessus du bloc rouge, en lieu et place de "glisser" contre le mur.



Avec les solutions à tous ces problèmes implémentés, nous obtenons un système de collisions très versatile, simple à personnaliser, et optimisé pour un monde de voxels.

2.5.2 Système de déplacement

Afin d'avoir un système de déplacement répondant à nos besoins, nous avons aussi recréé le système de déplacement de Unity, trop limité. Nous pouvons donc inclure dans le système d'héritage des entités des propriétés communes, comme la gravité, la friction, afin de garantir une unité parmi les entités sans avoir à modifier une même propriété à plusieurs endroits dans la scène.

Le système de déplacement est par ailleurs pairé avec le système de gestion des collisions, ce qui le rend plus performant. Par exemple, si le joueur se déplace à droite, il est inutile de calculer les collisions à sa gauche.

2.5.3 Custom network transform, network animator

Un autre aspect qui a été créé de manière customisée est le Network Animator de Mirror, qui permet de synchroniser les animations dans tous les clients.

Nous le substituons de par notre système de multijoueur pour les joueurs et entités, qui permet de synchroniser des énévenements (démarrer une animation, en arrêter une autre). Ce système est aussi utilisé localement afin d'avoir un *wrapper* plus facile d'utilisation autour du système d'animations de Unity.

2.6 Interface utilisateur et HUD

2.6.1 Inventaire

Dans IsoCraft Story, l'inventaire du joueur est composé de quatre lignes et neuf colonnes, qui peuvent chacune comporter un ou plusieurs objets. Une case ne peut comporter que des objets du même type.

Le nombre d'objets maximum est variable : un outil par exemple, est un objet contenant un script pour gérer sa durabilité par exemple. Il ne peut donc être stocké dans la même case qu'un autre outil, même s'ils sont du même type. Les autres objets peuvent être en général jusqu'à 32 dans la même case d'inventaire.

Le joueur récolte les blocs qu'il casse dans son inventaire, et les place depuis celui-ci. On remarque que la quatrième ligne de l'inventaire est reflétée en bas de l'écran : c'est la "hotbar" du joueur. Le joueur y place les objets qu'il veut facilement attraper en jeu. On peut cliquer sur les objets de l'inventaire pour les changer de case, et sélectionner un ou plusieurs objets d'une case en utilisant

différents boutons de la souris.

Dans le jeu, une fois que l'inventaire est fermé, le joueur peut sélectionner un objet parmi sa hotbar en déplaçant le curseur avec la molette de la souris. Sur l'image ci-dessus, on peut voir qu'un bloc de tas de planches de chêne est sélectionné. Si le joueur clique avec le bouton droit de sa souris, il pourra poser un de ces blocs dans le terrain, qui va être enlevé de son inventaire.

Si le joueur décide de récolter un bloc du terrain, il peut utiliser le clic gauche de sa souris et le bloc récolté va se rajouter dans son inventaire, en priorité dans la case sélectionnée, puis dans une case disponible. Le bloc peut aussi s'associer avec des blocs du même type s'il y en a dans l'inventaire.

Il est à noter que, en vertu d'une progression variée et d'un réalisme accru, le bloc cassé ne va pas forcément être celui récolté par le joueur. Nous reviendrons sur ce point lorsque nous parlerons des outils.

2.6.2 Chat

Le chat de IsoCraft story permet aux joeurs de communiquer in-game, sans microphone, mais également d'exécuter des commandes spécifiques, exécutées sur le serveur.

Lorsqu'un joueur envoie un message via le chat, le message va être envoyé au server, puis analysé :

— Si le message ne commence pas par un `/`, c'est un message normal qui va être envoyé à tous les autres joueurs.
— Si le message commence par un `/`, c'est alors une commande, qui ne va pas être envoyée aux autre joueurs mais plutôt va être exécutée

Quelques exemples de commandes sont `tick rate [nombre]`, qui permet de changer la vitesse du jeu, `tp`, qui permet de téléporter un joueur vers un endroit donné dans une couche donnée.

2.6.3 Tutoriel

Dans un jeu comme IsoCraft Story, il est crucial de guider les joueurs, en particulier au début, afin qu'ils ne se sentent pas perdus. Pour cela, nous avons adopté plusieurs méthodes destinées à enseigner les contrôles et à orienter les

joueurs tout en leur laissant une grande liberté d'action. Ce rapport détaille ces méthodes et leur mise en œuvre pour garantir une expérience de jeu fluide et immersive.

Pour éviter que le joueur ne se sente désorienté dès le début du jeu, nous avons intégré une série de mécanismes de guidage, d'abord sous forme d'un tutoriel, qui s'affiche en haut à droite de l'écran, dans un popup animé qui réagit aux actions du joueur.

Afin que le joueur ne soit pas perdu, le tutoriel est assez robuste. Par exemple, le joueur doit effectuer l'action expliquée dans le tutoriel pendant que le popup l'expliquant est actif. Dans le cas où le joueur aurait effectué l'action avant et l'ait comprise, il pourrait s'attendre à ce que le tutoriel ait quand même enregistré son action. C'est le cas, le popup reste juste actif plus longtemps pour que tous les types de joueurs aient compris l'action qui leur est apprise.

Enfin, l'état actuel du tutoriel est sauvegardé dans le fichier de sauvegarde du joueur, ce qui implique que tous les joueurs ont accès au tutoriel pour une partie donnée, pas seulement le premier arrivé, et si un joueur quitte le jeu pendant le tutoriel, celui-ci reprendra là où il s'est arrêté.

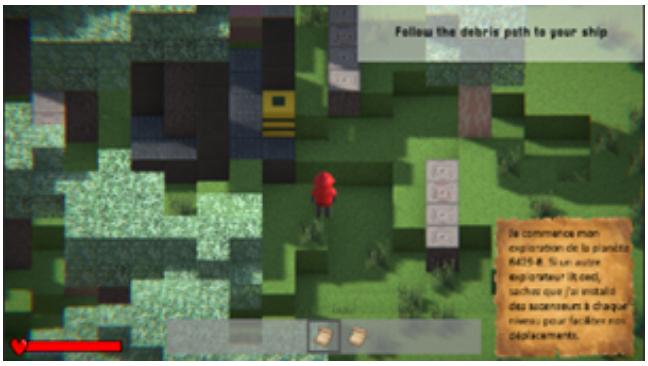
Après avoir expliqué les mécaniques de base au joueur, le tutoriel s'assure que le joueur les a bien enregistrées en continuant le tutoriel de manière interactive, aidant le joueur à se déplacer vers l'épave de son vaisseau. Le vaisseau sert ainsi de lieu de départ pour les explorations.

Bien que le joueur soit libre de ses actions après avoir atteint le vaisseau, nous avons implémenté des mécanismes de guidage subtils pour maintenir l'engagement et la direction générale sans restreindre la liberté du joueur :

Le joueur est fortement incité à explorer un étrange ascenseur situé près du vaisseau - étant l'un des seuls éléments distinctifs proches de la zone de départ. En prenant cet ascenseur, le joueur peut descendre dans la couche inférieure, ouvrant ainsi de nouvelles zones à explorer.

Mais le joueur a encore un long chemin devant lui avant de pouvoir emprunter cet ascenseur, car ce dernier est obstrué par des rochers qu'il lui faudra briser avec un outil adapté, lui apprenant les bases du système de fabrication dans un environnement simple et sans distractions.

Une fois que le joueur est capable de descendre dans la couche du dessous, il trouve des documents qui lui indiquent un chemin à suivre. Ces documents fournissent des indices sur les objectifs à atteindre et les secrets à découvrir. Le joueur est libre de suivre ces indications ou de choisir une autre voie.



Ainsi, IsoCraft Story est conçu pour offrir une grande liberté d'action aux joueurs. Après avoir reçu les premières instructions et indices, le joueur peut choisir de suivre l'objectif principal ou de s'engager dans d'autres activités. Cette structure ouverte permet aux joueurs de vivre une expérience personnalisée et de découvrir le jeu à leur propre rythme.

De plus, le seul élément semi-intrusif est le tutoriel au début. Or, celui-ci réagit à des actions courantes dans une partie normale. Les joueurs plus expérimentés pourront donc complètement ignorer ce tutoriel et celui-ci se terminera naturellement. Ainsi, tous les types de joueurs sont représentés et peuvent avoir une expérience de IsoCraft Story selon leurs besoins. Un joueur expérimenté pourrait par exemple vouloir directement commencer à accéder à la zone du dessous s'il a un objectif en tête, et il ne sera pas entravé par le tutoriel intuitif, son imagination poursuivant son libre cours.

Voici un exemple en image, avec le tutoriel indiquant une information au joueur en français :

2.6.4 Lang

Comme indiqué précédemment, le tutoriel était ici affiché en français. Nous avons commencé à implémenter la gestion de différentes langues en tant qu'amélioration secondaire. Pour le moment, seuls les noms des blocs et le tutoriel sont traduits, car traduire les menus est une tâche fastidieuse que nous n'avons pas encore entreprise.

La traduction fonctionne en récupérant les textes du jeu dans un fichier externe, sous un format bien particulier inventé pour l'occasion, afin de permettre une transcription ultérieure très rapide et en plus de langues.

Cette option de traduction représente très bien notre méthode de travail pour IsoCraft Story : à ce point, nous nous sommes principalement focalisés sur des aspects plus techniques du jeu, afin de rendre l'ajout de fonctionnalités très rapide et modifiant un minimum de code. Par exemple, nous avons commencé beaucoup d'ajouts supplémentaires à notre jeu, "au cas où", ce qui permet un développement de contenu efficace.

Ayant posé les baes du jeu et n'en restant plus à développer, le développement de IsoCraft Story est et restera exponentiel pendant le temps de développement que nous allons mettre à profit après le rendu de ce projet. Après avoir démontré nos capacités à développer des outils de développement, nous avons maintenant un moteur très puissant et versatile.

2.6.5 Réglages, sauvegarde des réglages

Les joueurs ont la possibilité de modifier leur expérience personnelle facilement, par le biais de réglages.

Ces réglages sont de deux natures différentes : d'abord les réglages par sauvegarde, gérés par le serveur. Ceux-ci comportent par exemple le nombre de joueurs maximum dans une partie, ou le délai entre deux sauvegardes automatiques.

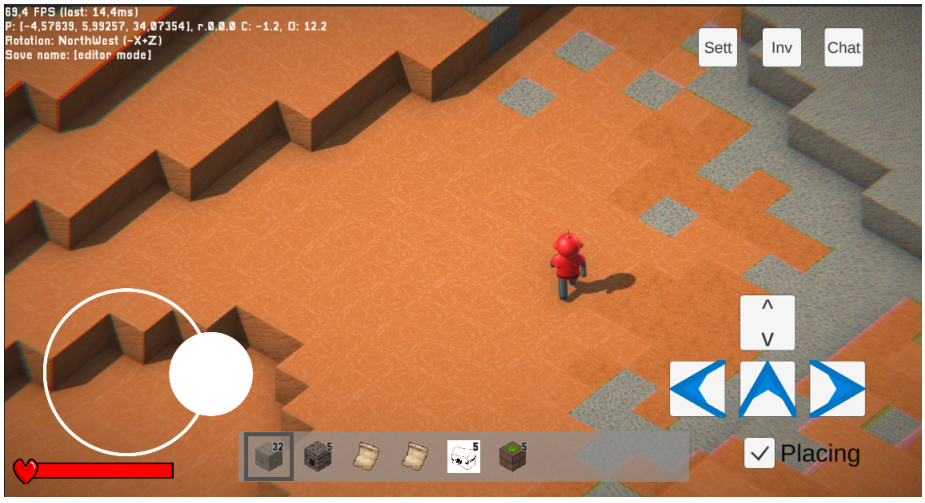
Les autres réglages sont par joueur, et concerent l'expérience utilisateur. Voici les réglages implémentés au moment de l'écriture de ces lignes :

— L'affichage d'un écran de debug, qui affiche des informations telles que la position du joueur, le nombre d'images par seconde
— La possibilité de limiter la qualité graphique en enlevant le post processing, et en limitant les faces des blocs dans les mesh des chunks (les faces entre blocs transparents ne sont pas affichées)
— Le nombre de mipmaps (différentes résolutions d'un même bloc utilisées pour parer à des artefacts lors de l'affichage d'un bloc lointain)
— La distance de rendu des chunks et des entités
— La langue
— Le volume de la musique
— Chaque touche peut être paramétrée (sauf pour afficher le menu de pause)

2.6.6 Prototype pour mobile

Afin de toucher une clientèle plus large, IsoCraft story existe également sur mobile, Android comme IOS. Les contrôles ont été adaptées pour la meilleur expérience mobile possible. Néanmoins, cet aspect du jeu en est encore à ses débuts, car certaines fonctionnalités, comme les structures par exemple, ne sont pas encore présentes.

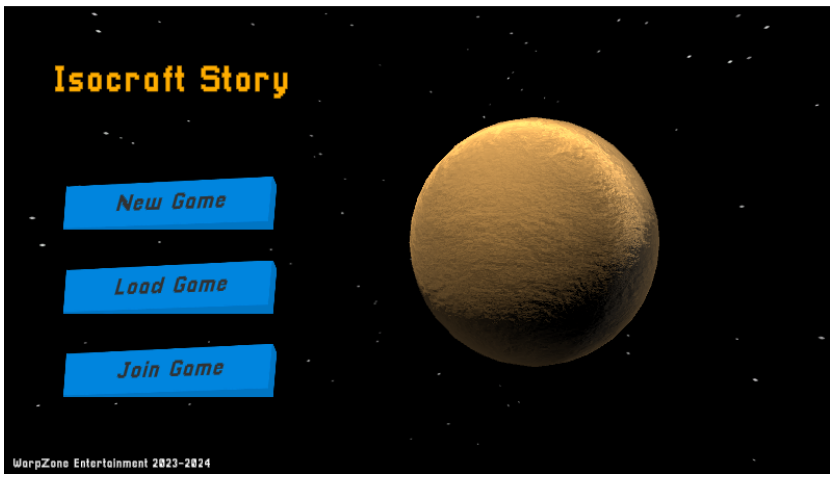
Le multijoueur est partiellement cross-platform, ce qui veut dire qu'un joueur mobile peut techniquement accéder à la même partie qu'un joueur PC. Il est partiellement fonctionnel, mais terminer cette facette du jeu n'est pas une de nos priorités pour le moment.



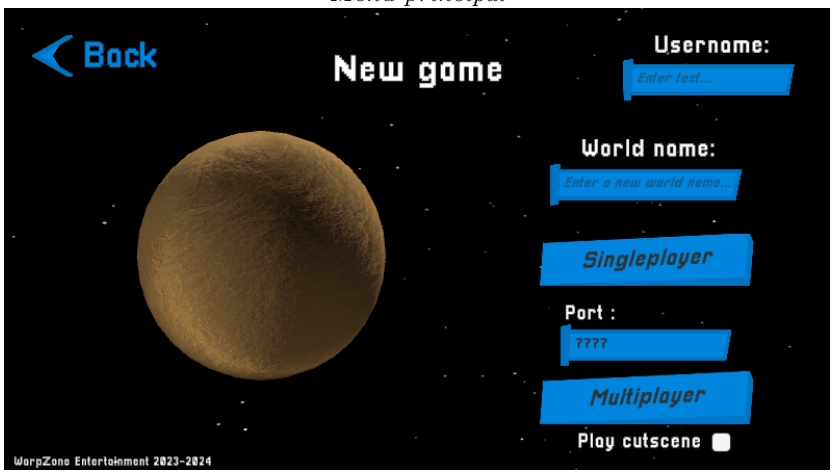
2.7 Menu principal

2.7.1 Menu principal

Voici deux captures d'écran dans le menu principal. On peut y voir 6429-B, la planète sur laquelle le joueur s'est écrasée. En fonction du menu sélectionné, diverses options sont disponibles :



Menu principal



Menu de création d'un fichier de sauvegarde

Le menu permet de créer une nouvelle partie (solo ou multijoueurs, mais comme dit précédemment ceci peut être changé). Lors de la création d'une nouvelle partie, la cinématique de début peut être lancée.

Il est aussi possible de lancer une partie existante en tant qu'hôte, ou d'en rejoindre une en utilisant un code de connection ou en saisissant manuellement l'adresse IP du serveur et le port sur lequel se connecter.

2.7.2 Animations du menu

Dans le menu principal, la caméra tourne autour de la planète sur un fond d'étoiles généré procéduralement. La planète est légèrement sur la droite, mais lors de la sélection d'un sous-menu la planète tourne plus vite et se décale vers la gauche.

Tous les boutons et menus se déplacent en une animation en fonction du menu, pour obtenir une transition fluide. Un fondu vers le noir est aussi présent lors de la lancée de la cinématique de début.

2.8 Block entities, item entities

Cette section revient sur le concept de block entity et de scripts associés aux blocs et objets de l'inventaire.

2.8.1 Outils et statistiques

Le joueur a possibilité de fabriquer divers outils, lui permettant de changer la manière et l'efficacité avec laquelle il interagit avec le monde.

Tout d'abord, chaque entité a un certain rayon d'attaque et d'interaction. Au-delà, il est impossible d'interagir avec le monde. Par défaut, par exemple, le joueur ne peut casser et poser de blocs qu'à moins de trois blocs de distance. Mais s'il fabrique un marteau, il peut atteindre quatre blocs.

En plus de la distance, d'autres statistiques sont modifiées. Par exemple, la puissance d'attaque. Mais certains outils sont plus ou moins efficaces pour détruire certains blocs. Par exemple, un marteau casse facilement de la pierre, mais

une hache coupe facilement le bois. Le joueur devra jongler entre ces différentes statistiques intuitives pour progresser dans l'aventure.

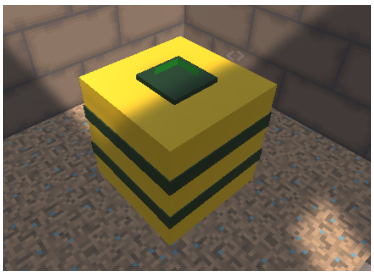
Nous avions évoqué précédemment le fait que certains blocs, une fois détruits, ne donnent pas forcément le même bloc au joueur. Par exemple, casser un bloc de grès à main nues donne une pierre, ou un bloc de sable donne de la poudre de sable. Mais un bloc pourra être donné au joueur s'il est détruit avec le bon outil. Si l'on reprend ce même exemple, casser un bloc de grès avec un marteau donne ce même bloc de grès, qui peut ensuite être transformé ("crafté") en plusieurs pierres. On a donc ici un meilleur rendement avec le bon outil.

De plus, les outils sont non seulement organisés en différentes catégories, mais ils ont un "rang", une efficacité au sein de leur groupe. Certains blocs nécessitent un certain rang d'outil pour être seulement brisés, ce qui guide le joueur à avancer dans l'aventure d'une autre manière, et à conquérir le monde hostile qui l'entoure.

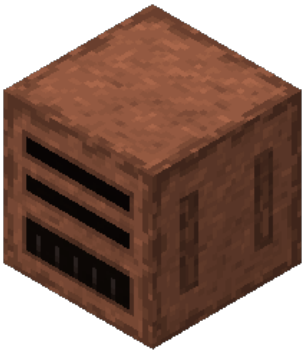
Parlons maintenant des types de block entities implémentés dans le jeu à ce jour.

Ruche (spawner)

Ascenseur vers le bas



Coffre (modèle 3D)



Four en grès

2.8.2 Spawners

Les "spawners" sont des blocs spéciaux qui peuvent générer des entités et ennemis. Ils ne peuvent en générer qu'une certaine quantité, au-delà ils sont bloqués. Tuer une entité générée par un spawner permet au spawner de générer une nouvelle entité.

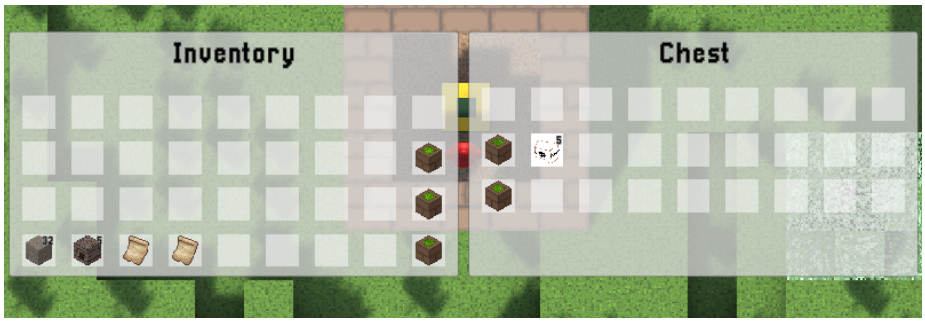
2.8.3 Ascenseurs

Les ascenseurs sont les blocs clés qui permettent au joueur de progresser dans l'aventure en passant d'une couche à une autre. Interagir avec un ascenseur - en cliquant dessus - fait changer le joueur de niveau, soit monter, soit descendre.

Le premier ascenseur du niveau 0 vers le niveau -1 est généré automatiquement dans le jeu, mais plus tard dans l'aventure le joueur sera en mesure de créer d'autres ascenseurs pour descendre plus bas ou en avoir des plus accessibles.

2.8.4 Coffres

Nous avons parlé des coffres précédemment, mais précisons ici leur fonctionnement. Ce sont des blocs capables de stocker des objets. Le joueur peut les trouver déjà générés dans le terrain, et y trouver des reliques, mais il peut aussi en fabriquer pour trier ses ressources plus facilement.



2.8.5 Utility blocks

Les "utility blocks" sont un type de block entities qui permettent de fabriquer des matériaux. Nous verrons leur comportement plus en détail lorsque nous parlerons du concept de crafting, mais pour ce qui est de leur importance dans cette catégorie, ils possèdent un inventaire de quelques cases qui peut interagir avec l'inventaire du joueur.

2.9 Lore

De base, le jeu IsoCraft Story devait nous permettre d'explorer une planète creuse composée d'une infinité de couches, toutes empilées les unes sur les autres, avec une difficulté réglée automatiquement par une augmentation du niveau des ennemis. Cependant, nous avons préféré adopter une approche plus personnalisée pour chaque couche afin de conférer différentes ambiances et une histoire riche à la planète. Cette approche a permis de développer une expérience de

jeu plus immersive et narrative. À la fin, nous avions plusieurs couches souterraines en tête, chacune avec ses propres caractéristiques uniques et une histoire détaillée.

2.9.1 Première Couche

Le Domaine des Oak Boka

— **Description**
À quelques centaines de mètres sous la surface, cette couche abrite les Oak Boka, un peuple aborigène vivant en parfaite harmonie avec leur environnement. Ils protègent leur communauté avec l'aide des Bopako, des oiseaux sacrés nichant autour des arbres de vie, qui relâchent des spores repoussant les Trogdors, des dinosaures aliens prédateurs.

— **Ambiance et Histoire**
Cette couche introduit une civilisation harmonieuse et respectueuse de la nature. Les joueurs découvrent les temples étranges et les maisons des Oak Boka, chargés de symboles anciens et de mystères enfouis.

— **Gameplay**
Cette zone paisible, avec peu d'ennemis, permet aux joueurs de comprendre les rudiments des mécaniques du jeu comme le crafting et l'exploration. Les ressources abondantes et les dangers limités offrent un environnement idéal pour les premiers apprentissages.

2.9.2 Deuxième Couche

Les Galeries de l'Oubli

— **Description**
Un labyrinthe de tunnels miniers et de laboratoires abandonnés, hanté par des Morbides, des créatures mutantes hostiles, et des Trogdors adultes.

— **Ambiance et Histoire**
Cette couche révèle les vestiges de technologies avancées et les traces d'expériences scientifiques sur l'immortalité, ajoutant une dimension de mystère scientifique et de danger.

— **Gameplay**
Les galeries de l'oubli augmentent la difficulté avec des ennemis plus

puissants et des environnements plus complexes. Les joueurs doivent naviguer dans des labyrinthes de tunnels et éviter des pièges, introduisant des mécaniques de survie avancées et de combat plus intensif.

2.9.3 Troisième Couche

Les Ruines de la Cité

— **Description**
Une ancienne cité en ruines.

— **Ambiance et Histoire**
Cette couche offre un aperçu d'une civilisation avancée aujourd'hui disparue. Les monuments et artefacts racontent l'histoire de cette civilisation.

— **Gameplay**
Dans cette couche, les joueurs doivent résoudre des énigmes pour avancer et découvrir des zones cachées. La collecte de ressources rares et la gestion de la résistance aux espièglerie des Nithrax demandent une planification stratégique et une exploration minutieuse.

2.9.4 Quatrième Couche

Les Hauts Quartiers de l'Ancienne Cité

— **Description**
Des quartiers patrouillés par d'anciens robots, contenant des cubes énergétiques et des technologies sophistiquées.

— **Ambiance et Histoire**
Cette couche explore la notion de survie numérique avec des esprits connectés à une réalité artificielle, ajoutant une réflexion philosophique sur l'immortalité.

— **Gameplay**
Les joueurs doivent éviter ou neutraliser des gardes immortels, en utilisant des techniques d'infiltration et de hacking. L'accès aux cubes énergétiques demande des compétences en résolution de puzzles technologiques et en combat tactique.

2.9.5 Cinquième Couche

La Cave des Profondeurs

— **Description**
Une cave abritant le Gardien Éternel, une créature immortelle maintenant une simulation pour une civilisation éteinte.

— **Ambiance et Histoire**
Cette couche représente le point culminant de l'exploration, où les joueurs doivent prendre des décisions cruciales affectant l'avenir de la simulation et de ses habitants.

— **Gameplay**
La confrontation avec le Gardien Éternel propose un défi ultime, combinant des éléments de combat intensif, de gestion des ressources et de prise de décision morale. Les joueurs doivent choisir entre libérer ou contenir le Gardien, influençant la fin du jeu.

Avec cette base, nous avons pu établir plusieurs histoires internes à la planète, telles que celles de Xar'lan, un explorateur audacieux, ou du Chef Oak Boka, chacune enrichissant le lore de 6429-B. Avec ces thèmes, nous pouvons alors construire un jeu se déroulant dans un monde cohérent et bien construit.

Cependant, bien qu'ayant développé le lore et l'histoire de la planète, nous n'avons pas encore pu implémenter certaines couches et certains mobs. Mais, comme dit plus haut, nous sommes depuis quelques temps dans la phase de développement accéléré, car nous avons les bases du moteur du jeu et il ne reste plus qu'à rajouter du contenu. Dans une semaine, qui sait ? nous pourrions avoir terminé.

2.10 Modeling

2.10.1 Apprentissage du Modeling

La première étape dans la création des modèles 3D pour IsoCraft Story a été l'apprentissage du modeling. Il a été essentiel de maîtriser les bases du modeling 3D pour créer des modèles fonctionnels. Les membres de l'équipe chargé de la création de ceux-ci ont suivi divers tutoriels en ligne sur la plateforme YouTube.

2.10.2 Croquis des Modèles 3D

Pour simplifier le processus de modélisation, la création de croquis détaillés des modèles s'est avérée indispensable. Ces croquis ont été réalisés à main levée ou sur Paint pour définir les formes et les détails nécessaires. Cette étape de préparation a permis de clarifier les intentions artistiques et de guider efficacement la modélisation 3D.

2.10.3 Optimisation des Modèles

L'optimisation des modèles a été une étape cruciale pour garantir des performances fluide en jeu, surtout sur des configurations matérielles variées. Les modèles low poly ont été créés en réduisant le nombre de polygônes sans compromettre l'intégrité visuelle.

Des techniques de réduction de polygones ont été appliquées à l'aide du logiciel Blender, tout en maintenant un équilibre entre qualité et performance. Des textures simples et efficaces ont été utilisées pour minimiser la charge sur le processeur graphique.

2.10.4 Implémentation des Animations pour les mobs

Les animations ont joué un rôle crucial pour donner vie aux personnages et objets dans IsoCraft Story. Unity a été utilisé pour animer les modèles, couvrant les mouvements de base comme la marche et les attaques.

Ainsi, La création des modèles 3D pour IsoCraft Story a été un processus intéressant nécessitant une combinaison de compétences techniques et créatives. En suivant les étapes décrites dans ce rapport, nous avons pu garantir que les modèles 3D enrichissent l'expérience de jeu tout en maintenant des performances optimales.

L'intégration réussie des animations et l'optimisation des modèles ont assuré que IsoCraft Story offrirait une expérience de jeu fluide aux joueurs.

2.11 Crafting

2.11.1 Utility blocks

Revenons sur les "utility blocks". Ce sont des blocs clés dans la progression du joueur, car ils permettent d'assembler les ressources trouvées sur la planète et d'en faire des objets plus avancés.

La liste de tous les schémas est présente dans le code du jeu, associant une ou plusieurs ressources créées à partir d'autres ressources. Par exemple, le joueur peut fabriquer un marteau en grès avec un morceau de grès, un bâton et de la ficelle.

Chaque block utility peut fabriquer un certain nombre de ressources, associé à son type de bloc. Par exemple, un établi en grès ne permettra pas de faire du travail de minutie autant qu'un en acier, mais ce dernier sera difficile à fabriquer, demandant au préalable un block entity capable de le fabriquer et les ressources demandées. Cela constituera le coeur de la progression du joueur.

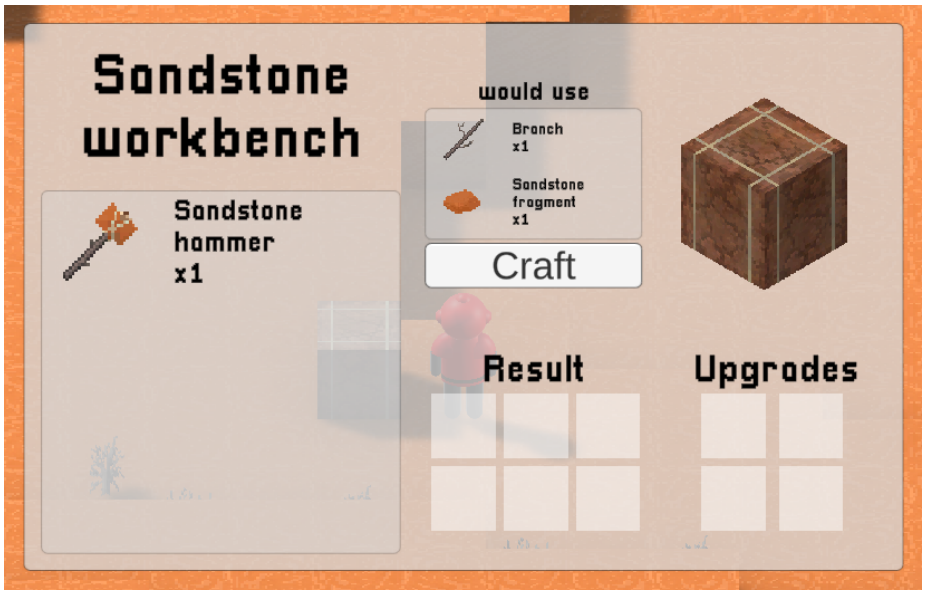
Lors de la fabrication d'un objet, il est possible que d'autres produits secondaires soient créés. Par exemple, lors de la coupe d'une bûche en planches, des branches peuvent être crées. Pour cela, le système de fabrication de ressources que nous avons implémenté permet la création de produits selon une certaine probabilité.

Il est donc possible que la fabrication d'une ressource ne soit pas toujours garantie, ou qu'un produit secondaire soit généré en quantité variable. Cela augmente le réalisme du système de fabrication par rapport à d'autres jeux, comme Minecraft, dont nous nous sommes inspirés.

Un objet ne peut être fabriqué que si le joueur a les matériaux requis. lorsqu'il est fabriqué, il est ajouté à l'inventaire du block entity. Le joueur peut ensuite cliquer sur l'objet dans l'inventaire, ce qui ouvrira son inventaire et lui permettra d'ajouter cet objet aux objets qu'il possède déjà.

Puisque l'utility block est un block entity, son contenu est publié pour tous dans le monde. Cela veut dire par exemple que quelqu'un peut fabriquer un objet, mais que quelqu'un d'autre puisse le prendre à sa place, mais aussi que les block entities peuvent servir de stockage lorsqu'aucun autre moyen de stockage n'est disponible.

Voici un exemple d'interface d'utility block :



Le joueur sait quels objets il peut créer en utilisant la liste déroulante sur la gauche. On y trouve les objets à fabriquer ainsi que le nombre et la probabilité de les fabriquer.

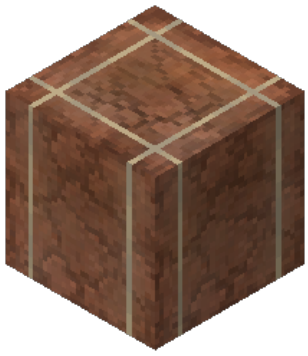
Lors de la sélection d'un de ces éléments, il devient possible de voir les ressources nécessaires à la fabrication de cette ressource et de cliquer sur le bouton Craft pour utiliser les objets nécessaires et fabriquer l'objet.

Le ou les objets fabriqués vont alors être placés dans les cases d'inventaire nommées "Result".

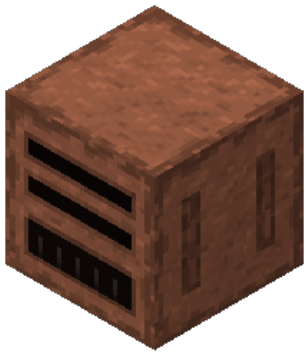
Les cases "Upgrades" ne sont pas utilisées pour le moment dans le contenu, mais pourront l'être par la suite si besoin.

2.11.2 Exemples de utility blocks

Voici quelques utility blocks pour illustrer nos propos : à gauche un établi en grès, à droite un four en grès. Ces blocs ne sont pas très puissants et permettent seulement la création d'outils rudimentaires, mais ils sont importants dans l'aventure car essentiels à la progression toute entière.



Etabli en grès



Four en grès

2.12 Musique et sound design

2.12.1 Musique dans les menus

Dans le menu du jeu, deux musiques se jouent simultanément. Ce sont les mêmes accords et mêmes tonalités, mais l'une d'elles est plus riche en instruments et harmoniques.

Bien sûr, les deux musiques ne sont pas actives tout le temps : la musique la plus calme, la plus simple, se joue dans le menu principal, et la musique plus complète se joue dans les sous-menus.

Tout l'intérêt de jouer les deux musiques prend son send dans le fait que, si l'on met le volume de l'une des musiques à zéro pendant que l'autre est à 100%, lors du changement de menu la progression dans la musique reste la même pour l'autre musique et il devient alors possible de mettre le volume de la première musique à 100% et celui de l'autre à zéro.

L'effet est encore plus intéressant et apprécié en effectuant un fondu entre les

deux musiques, en augmentant doucement le volume de l'une tout en diminuant celui de l'autre. On obtient alors une totale symbiose entre les musiques et les animations de changement de menu.

2.12.2 Système de musique dans le jeu

Dans le jeu, le système de musique est complètement différent. Le thème du jeu est la libre créativité, aussi les musiques se font discrètes. On observe une pause de 30s à 5min entre chaque musique pour laisser au joueur le temps de se forger un lien avec le jeu, car il est seul sur une planète exotique et inconnue, à explorer et progresser.

Tout à coup une musique légère commence, et elle ne fait que renforcer ce lien entre le joueur et le jeu, elle a pour but de ne pas prendre trop de place dans le jeu, et de laisser le joueur libre cours à ses pensées créatives.

Cependant, IsoCraft Story possède quelques zones clés à l'aventure. La plus importante est le site du crash de son vaisseau. On notera aussi les emplacements de maisons Oak Boka, à caractère rustique et indigène, ainsi que le laboratoire. Dans ces endroits uniques, certaines musiques dédiées se lancent en continu. Elles servent à renforcer une sensation unique et propre au lieu, ou presque à servir de refuge contre l'immensité du vide en dehors de ces structures isolées.

Lorsque le joueur quitte ces zones, il se rend à nouveau compte du fait qu'il est seul sur cette planète, isolé de tous et cherchant vainement à reconstruire son vaisseau. Cela permet au joueur de mieux s'identifier avec son personnage.

Bien sûr, le fait d'être plusieurs à jouer au jeu en même temps limite certes cette expérience, mais cela n'est pas un mal, au contraire.

Cela permet de clairement différencier deux aspects possibles du jeu : se perdre sur une planète exotique, seul à reconstruire un vaisseau et partir, ou alors construire une civilisation avec l'aide de ses coéquipiers et ne jamais quitter cette planète qui donne libre cours à notre imagination.

2.12.3 Music design

Les musiques reflètent le caractère de la couche dans laquelle ils sont joués. Elles sont composées en utilisant le logiciel Maschine 2.

— Le niveau 0 est aride, hostile, et le personnage est constamment rappelé son histoire et le fait qu'il est bel et bien coincé sur cette planète. Les musiques sont caractérisées par des mélodies au piano, violons mélancoliques.

— Le niveau -1 est luxuriant, vert et riche, mais le joueur sait qu'il est toujours seul. La présence d'autres entités le ravive cependant, et les musiques dans les maisons Oak Boka sont joyeuses.

— Le niveau -2 est caractérisé par la présence d'un laboratoire abandonné, où se joue une musique riche en harmoniques et du passé. Le niveau est cependant sombre, et les mélodies tendent vers un son plus dur et plus sec, mais aussi plus grave.

— Le niveau -3 est plus sombre, et on n'y trouve pas une grande quantité de vie. Les musiques sont oppressantes, distordues, parfois reprenant des thèmes des niveaux précédents.

— N'ayant pas avancé le développement sur les niveaux suivants, aucune musique n'a encore été composée pour ces couches. Mais nous ne nous projettions pas sur un nombre de couche précis, donc la composition s'est arrêtée là pour le moment.

2.12.4 SFX ?

Pour le moment, nous n'avons pas eu le temps d'implémenter d'effets sonores dans le jeu, mais il est probable que nous ajoutions cette fonctionnalité très rapidement.

Nous avions prévu de jouer des sons lors de la découverte d'une nouvelle couche, ou lors d'actions telles de détruire un bloc, marcher sur des blocs, frapper un ennemi, mais nous avons manqué de temps ou ne nous sommes pas assez focalisés sur cet aspect du jeu.

2.12.5 Cinématique du début

La cinématique du début du jeu est découpée en plusieurs parties, et la musique est en accord avec celles-ci :

— Tout d'abord, le joueur quitte sa planète d'origine. Il est joyeux mais calme, et les harmoniques de violons caractérisent sa plénitude.

— Ensuite, le joueur croise une barrière d'astéroïdes. Les accords commencent à être dissonnants en même temps que le joueur devient anxieux.

— Le vaisseau est balloté et frappé par des astéroïdes. Le thème d'abord joué par les violons s'intensifie, est repris par plus d'instruments. On remarque que les notes allaient de grave à aigu, mais commencent à être jouées en même temps par d'autres instruments qui descendent en tonalité. Ceci caractérise le fait que le joueur sait qu'il va s'écraser, il se rapproche d'un accident certain et son vaisseau pert de l'altitude.

— Le vaisseau du joueur est frappé, un orgue rejoint les instruments qui descendent, ainsi que des voix représentant les souvenirs du joueur. Un set de batterie accélère la cadence pour symboliser les émotions grandissantes du joueur.

— L'orgue va devenir l'instrument principalement utilisé pendant le reste de la cinématique. Il représente la fatalité. Le vaisseau du joueur se rapproche du sol, puis est frappé par la fatalité / un dernier astéroïde qui le fait perdre le contrôle puis s'écraser.

— Le vaisseau s'est écrasé, le bruit de l'explosion couvre la musique qui s'est d'ailleurs arrêté un court instant. Cela figure le fait que le vaisseau, la vie du joueur, son seul moyen de repartir vers la sécurité et les siens, s'est écrasé. L'orgue est le seul instrument qui recommence à jouer, en même temps que le joueur réalise qu'il ne peut plus repartir.

— On voit le joueur parachuté sur la planète, seul au milieu des dunes. La fatalité, ou l'orgue, continue de jouer de plus en plus fort.

— Fondu vers le noir. Le joueur prend le contrôle et le jeu commence. Il n'y a pas de musique, le joueur est seul.

2.13 Cinématique

Afin de raconter l'histoire du jeu au joueur, des cinématiques sont intégrées au jeu, par exemple au moment de la création d'une nouvelle partie, une cinématique est jouée expliquant au joueur d'où il vient et pourquoi il se retrouve sur une autre planète.

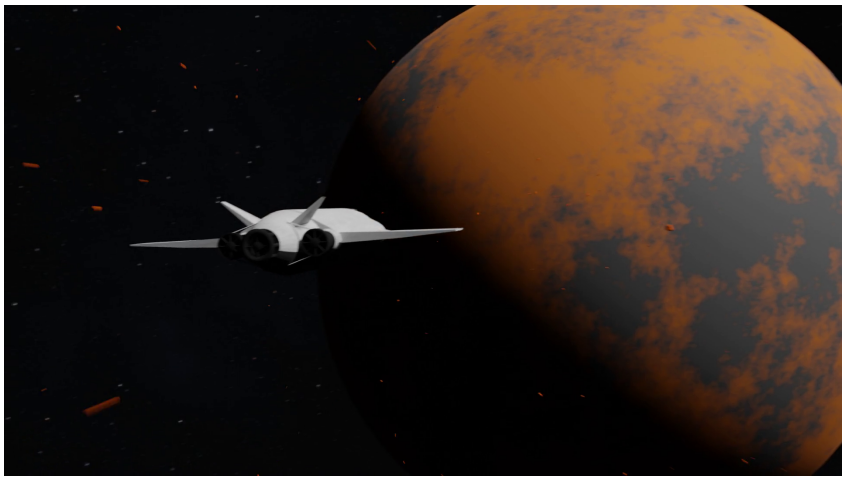
Ces cinématiques, réalisées sur Blender, n'utilisent aucune asset ou modèle 3D d'internet, car tout est modélise et simulé par les membres de notre groupe. Elles pour objectif d'être d'un style plutôt photo-réaliste, en contraste avec le

jeu sous forme de blocs.

La cinématique jouée au début d'une partie comporte pour le moment trois scènes :

— La première, où l'on peut voir le décollage de la fusée du joueur.

— La deuxième, où le joueur voyage dans l'espace, et rencontre un nuage d'astéroïdes, qui abîme son vaisseau.



— La dernière, où son vaisseau est très abîmé. Le joueur est alors éjecté, et son vaisseau s'écrase sur la planète 6429-B.

Ensuite, la partie se lance, et le joueur peut retrouver les débris de son vaisseau proche de lui, lui montrant que ce qu'il a vu dans la cinématique est bel et bien son histoire.

Une autre cinématique se jouera à la fin du jeu, ou l'on pourra voir le joueur quitter la planète.

Les modèles 3D sont créés et animés à la main.

Les fumées des flammes sont simulées grâce aux "fluid bake" de Blender. Cela permet un meilleur réalisme, au coût d'une animation longue à calculer pour l'ordinateur : en effet, les physiques de ces fluides sont pré-calculés initialement, pour être ensuite rendus lors de l'animation.

Plusieurs dizaines d'heures de calculs au total seront nécessaires pour que l'ordinateur produise les cinématiques finales.

2.14 Scripts python

Cette section est placée en dernier, car elle porte sur des outils transversaux que nous avons développé pour nous permettre d'être plus efficaces lors du développement du jeu.

Ces scripts adressent des opérations plus ou moins élémentaires mais surtout répétitives ou difficile à effectuer à la main, et ont été codés en Python.

2.14.1 Code Reviewer

Ce fichier est utilisé pour nettoyer notre code. Il va modifier le contenu des fichiers de code pour respecter certains standards de C# : mettre un saut de ligne après une accolade fermée, enlever les espaces à la fin des lignes et les sauts de ligne qui se suivent...

En plus de modifier notre code pour certaines opérations élémentaires, ce script va modifier des noms de fichiers et encodages pour simplifier leur utilisation par d'autres scripts : par exemple, comme nous le verrons bientôt, les fichiers de textures utilisés devraient respecter un certain format, par exemple être en minuscules, de taille 32x32... ce script va donc modifier ces images.

La partie la plus importante de cet utilitaire est la phase de *feedback* : après avoir lu tout notre code, le script va être capable de donner des indications pour nous rendre plus efficaces. Les aides qui composent la liste suivante sont associées à leur position dans le fichier correspondant, pour nous permettre de les adresser au plus vite. L'utilitaire nous propose d'autres méthodes d'optimisation, mais en voici les principales :

— **Méthodes lentes**
L'utilisation de certaines méthodes est dépréciée, car elles utilisent beaucoup de ressources. Il faut donc limiter leur utilisation dans les contextes critiques. Le script les identifie et nous les indique.

— **Méthodes problématiques**
Certaines méthodes ont un fonctionnement qui repose sur l'organisation de notre projet, et en cas de modification de ce dernier, elles pourraient cesser de fonctionner. Leur utilisation est donc dépréciée.

— **Lignes longues**
Des lignes trop longues diminuent la lisibilité, aussi l'utilitaire nous les indique.

— **Respect des conventions**
Des conventions ont été mises en place : espace avant et après un opérateur, espace après mais pas avant une virgule... le code nous aide à identifier ces problèmes.

— **TODO**
Nous plaçons dans le code des indications de fonctionnalités à modifier, ajouter ou enlever. Or, ces indications peuvent être perdues dans le long terme, c'est pourquoi l'utilitaire nous les indique et nous permet de ne jamais laisser une fonctionnalité à moitié terminée et d'avoir un code propre.

2.14.2 Blocks Parser

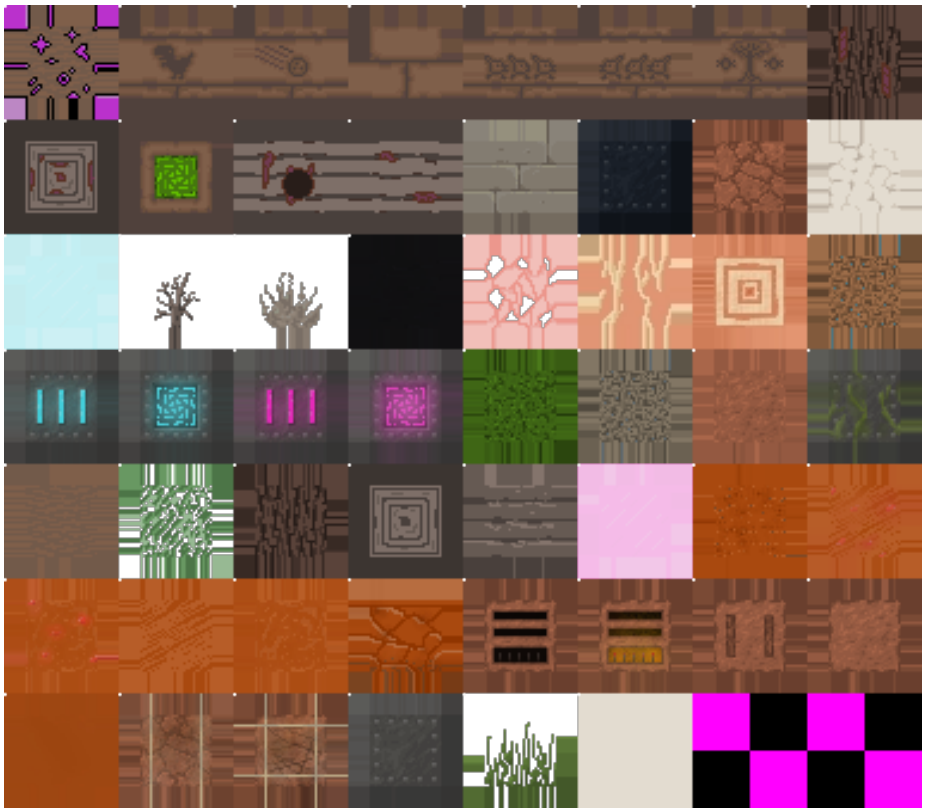
Ce fichier est utilisé par d'autres utilitaires, et il va lire le contenu de certains fichiers du jeu pour en tirer des informations, notamment des informations sur les blocs.

Par exemple, prenons l'établi en grès. Cet établi a un certain ID, et il a plusieurs textures : une pour la face du dessus, une autre pour les faces de côté et une dernière pour la face du dessous. De plus, c'est un block entity, qui peut être tourné, et il nécessite un outil de destruction de pierres de rang au moins 1 pour être récupéré.

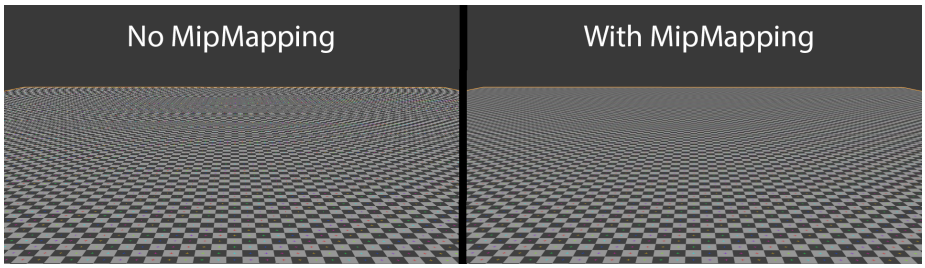
Toutes ces informations sont utiles dans le jeu, mais elles sont aussi utilisées par divers scripts et utilitaires. Ce script va donc récupérer toutes les informations cités ci-avant, pour tous les blocs, ainsi que les textures correspondant à chacune des faces des blocs. Le résultat est une structure de donnée en Python utilisable facilement par certains scripts.

2.14.3 Texmap Maker

Nous avons utilisé l'image ci-dessous précédemment. C'est une texture qui comprend toutes les textures des différents blocs, appelée tex atlas, ou texmap. Nous n'avons pas créé cette texture à la main, c'est le programme `Texmap maker` qui l'a créé.



Vous pouvez remarquer que les textures dans ce tex atlas semblent étirées. Les textures apparaissent de cette manière à cause d'une technique appelée "mipmapping". Cette technique essaye de réduire les artefacts dus à l'affichage de blocs assez loin. Ici une comparaison d'une même scène avec et sans mipmapping :



Le mipmapping est très utile, mais, dans notre cas, des problèmes supplé-

mentaires apparaissent. Ils proviennent du fonctionnement intrinsèque du mipmapping :

Lorsque la technique de mipmapping est utilisée, au lieu d'une seule texture, plusieurs sont stockées en mémoire : la texture originale, puis la même texture redimensionnée de plus en plus petit. Lors de l'affichage d'une texture, le moteur de rendu va choisir quelle texure afficher : si la face est proche de la caméra, la texture originale va être utilisée, et si la face est lointaine, un mipmap de plus basse résolution sera utilisé à la place.

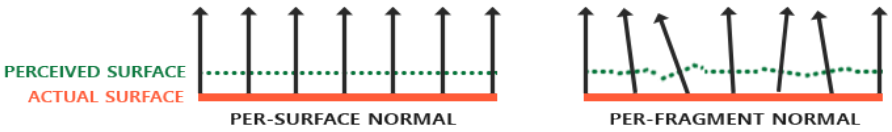
Le problème vient du fait que, pour générer les images de plus basse qualité, le moteur va "mélanger" les pixels pour ne pas perdre d'information. Par exemple, lors de la redimension d'une texture de 2x2 pixels en 1x1 pixel, le pixel final va être constitué de la moyenne des quatre pixels originaux.

Dans le cas d'un tex atlas, puisque les textures des blocs sont côte à côte, lors de la redimension les pixels de certains blocs vont "déborder" dans la texture d'un autre bloc. Dès lors, d'autre artefacts vont apparaître pour les blocs lointains ou regardés d'un angle très aigu.

La solution envisagée est une technique nommée "padding" : on rajoute une marge autour de chaque bloc. De cette manière, lors du calcul de moyenne des couleurs de pixels la moyenne est calculée avec les couleurs du même bloc et cela permet de n'avoir pas ou presque pas d'impact visuel.

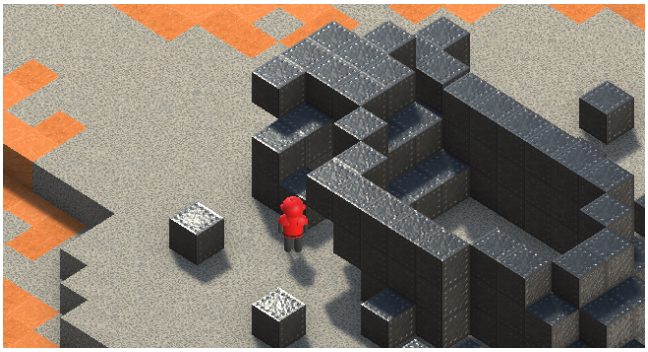
2.14.4 Normal maps

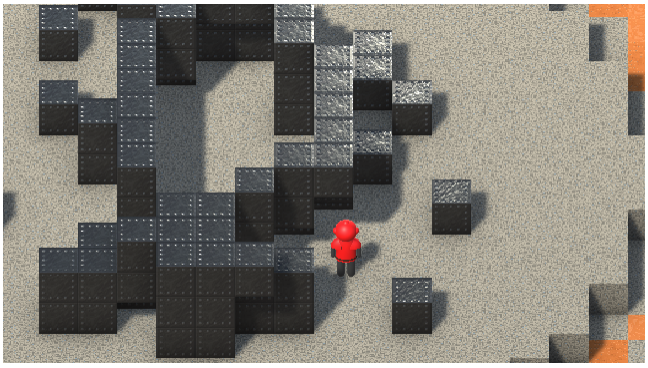
Les blocs n'ont pas seulement une texture 2D, ils ont aussi une texture permettant de donner une "orientation" de la texture pour chaque pixel, c'est-à-dire, pour chaque pixel, l'orientation du vecteur normal à la face. De manière classique, le vecteur normal pointe vers l'extérieur et perpendiculairement à la face, mais en utilisant une texture pour le spécifier explicitement, nous pouvons obtenir des effets de lumière intéressants.



En effet, lors du calcul de l'illumination d'un certain bloc, et de manière

ultime à l'allumage des pixels de l'écran de l'utilisateur, le moteur de rendu fait "rebondir" la lumière d'une source de lumière vers la caméra en utilisant le vecteur normal des faces. En spécifiant un vecteur normal personnalisé, nous pouvons par exemple obtenir cet effet de lumière sur des blocs de métal :





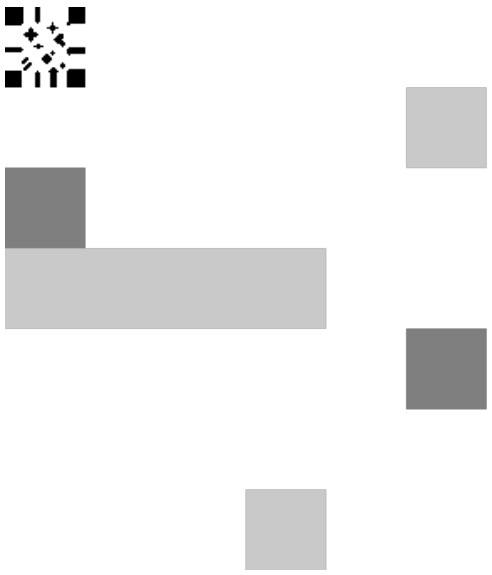
Voici la normal map à ce jour, générée par le script correspondant à partir du tex atlas :

2.14.5 Materials

Une autre modification, là aussi générée par un script, porte sur les *matériaux* des blocs. Un matériel est constitué de paramètres à donner au shader - un processus utilisé pour calculer les pixels à afficher à l'écran. Ces informations incluent par exemple l'émissivité d'une texture, ou sa réflexivité.

Nous disposons d'un script qui va générer, pour chaque bloc - soit pour chaque partie du tex atlas - , ces propriétés à partir d'un fichier que nous pouvons préparer rapidement, avec une syntaxe spéciale nous permettant une grande variété de possibilités, comme par exemple générer des options uniques par pixel du tex atlas.

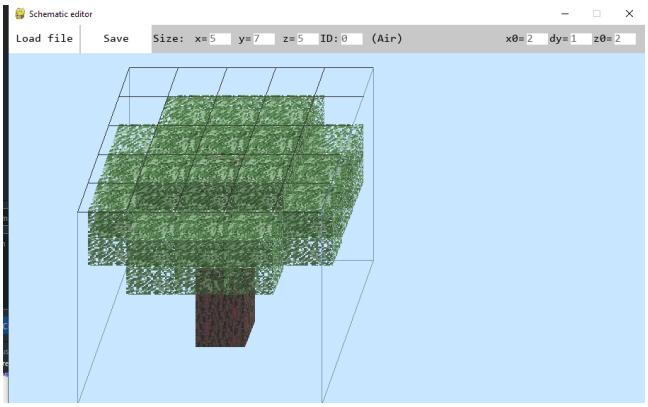
Le résultat est une autre texture qui, bien que peu compréhensible par des humains, permet au shader de savoir clairement la réflexivité de chaque bloc, si c'est un métal, et s'il est lisse ou non. Tout cela nous donne plus de réalisme et de variété pour les blocs : nous pouvons alors très bien créer de la terre humide, ou même du verre dépoli. Voici la texture générée par ce script :



2.14.6 Structure Editor

Nous avons évoqué au début le concept de structures : des assemblages prédéfinis de blocs à placer dans le monde à des endroits aléatoires ou donnés.

Nous avons développé un outil nous permettant de modifier facilement ces structures et de les exporter dans un format compréhensible par le jeu. Cet outil avait déjà été développé à la première soutenance, mais il a grandement évolué, par exemple il est maintenant capable d'afficher les textures des blocs en trois dimensions et non pas seulement des couleurs.



2.14.7 3D Blocks Sprites

Nous avons déjà présenté quelques blocs sous cette forme :

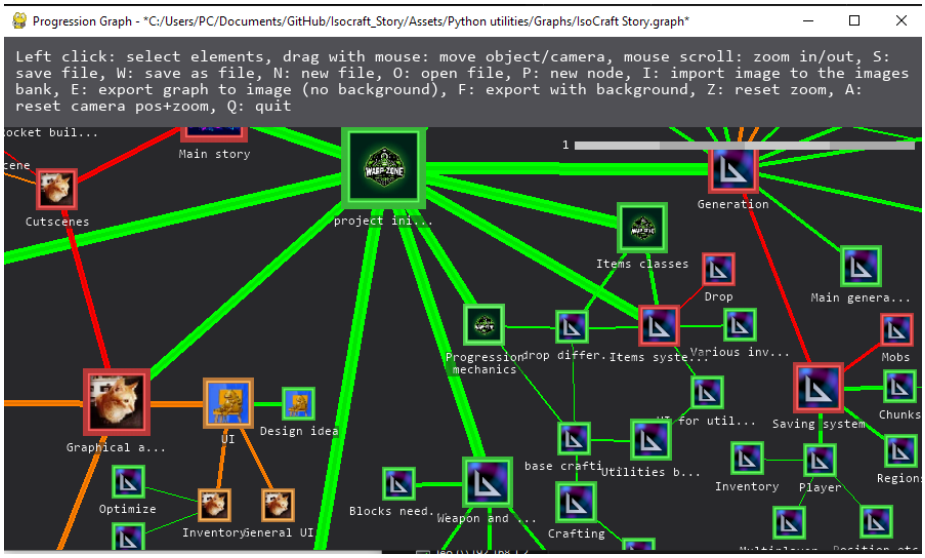
Ces blocs n'ont, là aussi, pas été générés à la main, ils sont rendus en trois dimensions par ce script. Cela nous permet alors d'avoir une représentation en items des blocs du terrain et de les afficher dans l'inventaire.

Le script utilise une technique de mapping fondée sur des calculs matriciels pour une efficacité maximale, et est un très bon exemple de l'utilité du parser de blocs : l'on peut être certain que les blocs vont utiliser les mêmes textures que celles utilisées dans le jeu.

2.14.8 Progression Graph

Afin de nous aider à nous rendre compte de notre progression relative au produit fini, nous avons développé cet outil de graphe de progression. Ce dernier permet de créer et relier des points, leur associer des images et textes, lesquels points ont un état d'avancement.

Cela nous permet par exemple de nous associer des tâches, ou encore de planifier l'avancement et la progression du joueur dans le jeu. Cet outil est utile ailleurs que dans notre jeu, autant nous l'avons publié et il nous est utile dans d'autres situations.

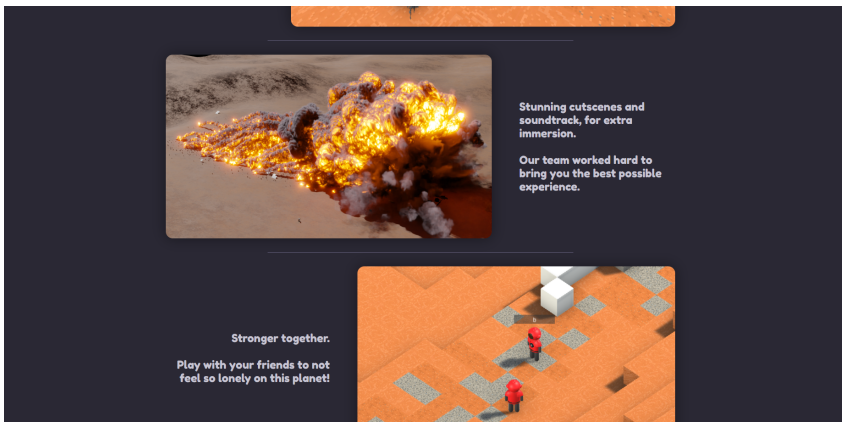

Progression Graph - *C:/Users/PC/Documents/GitHub/Isocraft_Story/Assets/Python utilities/Graphs/IsoCraft Story.graph*
Left click: select elements, drag with mouse: move object/camera, mouse scroll: zoom in/out, S: save file, W: save as file, N: new file, O: open file, P: new node, I: import image to the images bank, E: export graph to image (no background), F: export with background, Z: reset zoom, A: reset camera pos+zoom, Q: quit
Main story
Cutscenes
project ini...
Generation
Items classes
Drop
Main genera...
Progression mechanics
Drop differ...
Items syste...
Various inv...
Mobs
Chunks
Saving system
Graphical a...
UI
Design idea
Optimize
Inventory
General UI
Blocks need...
Weapon and ...
base crafti...
Utilities b...
Crafting
Inventory
Player

3 Site web

Le site web est terminé. Il comporte plusieurs pages :

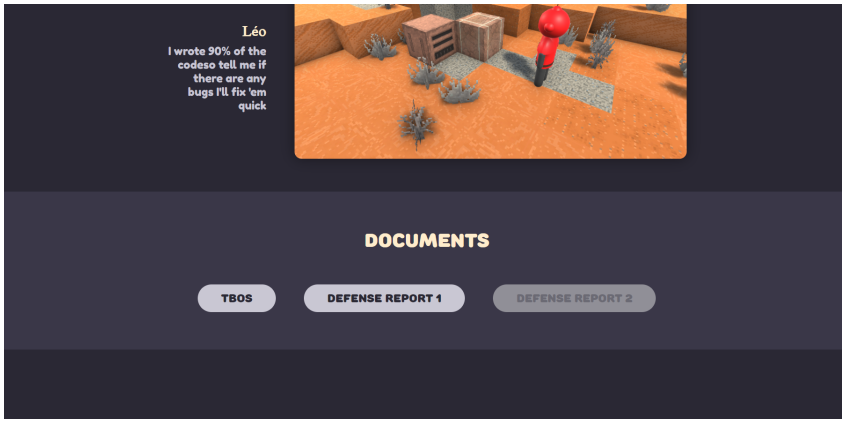
— La page d'accueil commence par une vue déroulante d'images panoramiques du jeu.

Plus bas, on peut trouver une liste d'aspects du jeu résumés : l'histoire, la vue isométrique, le système de craft et de progression, les cinématiques et le multijoueurs.



Enfin, vers le bas de la page on trouve des liens de téléchargements pour le jeu, une version allégée, et un lien vers le code source.

— La page "A propos" comporte une courte description des membres du groupe, et on peut y trouver un lien vers le cahier des charges et les deux rapports de soutenances.



Le lien vers le site web peut être trouvé en annexe et dans le repo git du projet.

4 Avances et retards

Voici le tableau d'avancement présenté lors du dernier rapport :

Soutenance	**Méthodo**	**1ère**	**Mi-mai**	**Dernière**
Génération / map	50%	75%	100%	100%
Complexité des voxels	0%	20%	X%	X%
Block entities	0%	0%	100%	100%
Histoire / lore	50%	75%	90%	100%
Design des couches	30%	60%	80%	100%
Progression, ressources	20%	50%	80%	100%
Crafting, alchimie	0%	0%	30%	100%
Boss, end game	0%	20%	40%	100%
Ascenseurs, liens niveaux	0%	0%	90%	100%
PVP, combat	0%	10%	80%	100%
Spawn des mobs	0%	0%	90%	100%
Items	0%	0%	50%	100%
Modélisation	10%	30%	50%	100%
Interface/design	30%	60%	75%	100%
Cinématiques	50%	75%	85%	100%
Balancing, playtest	0%	25%	50%	100%
Musiques	10%	40%	70%	100%
FX	0%	0%	20%	100%
Post processing, review	10%	20%	50%	100%
Animations	0%	0%	X%	X%
Multijoueurs / networking	80%	90%	100%	100%
IA	0%	20%	70%	100%
Site internet	0%	50%	80%	100%

"X%" signifie que la direction correspondante n'est pas réellement nécessaire, mais peut-être une piste d'avancement.

Considérons les catégories qui nécessitent des explications, ou où le déroulement ne s'est pas fait comme prévu :

— **Complexité des voxels**
Cette partie se réfère au fait que la carte doit pouvoir contenir des modèles 3D. Cette fonctionnalité est implémentée, comme expliqué plus tôt.

— **Design des couches**
Bien que nous ayons une claire idée de comment remplir les couches les

plus profondes (voir "Lore"), et que nous ayons la possibilité de les implémenter très rapidement, nous n'avons pas encore ajouté celles-ci au jeu. Comme expliqué plus tôt, notre processus de développement est accéléré maintenant que nous disposons des outils nécessaires, et il est probable que cette situation change au cours des quelques jours qui vont suivre.

— **Progression, ressources**
Le système de progression et de ressources est implémenté et fonctionnel.

Pour ce qui est du contenu lié à ce système, nous avons une idée claire du contenu du jeu, mais celui-ci n'est pas encore satisfaisant de notre point de vue : bien que la progression soit suffisante pour un jeu complet, elle manque encore de rigueur et certains points sont trop hâtés : par exemple, à ce jour l'âge de pierre occupe la majeure partie du temps de jeu, car tout va très vite après. L'implémentation de couches supplémentaires pourrait nous permettre d'étoffer cette progression.

— **Boss, end game**
Pour le moment, nous n'avons pas implémenté la fin où le joueur reconstruit son vaisseau. Bien qu'il est possible d'arguer que le principe du jeu n'est pas de le terminer, mais de découvrir l'environnement et de laisser libre cours à son imagination et son esprit d'exploration, nous nous devons de reconnaître que nous n'avons pas respecté le planning sur ce point.

— **Items**
Cette catégorie concerne la gestion des objets dans l'inventaire du joueur, et leurs différentes interactions - par exemple, les outils et armes. Ce point a correctement été implémenté conformément au planning.

— **Modélisation**
Nous sommes en retard sur ce point, car plusieurs mobs restent à être modélisés.

— **FX**
Comme énoncé précédemment, nous n'avons pas ajouté d'effets sonores autres que le système complet de musique.

Les autres points du tableau d'avancement sont réalisés avec succès conformément à la planification.

Nous pouvons tirer de ce tableau que, bien que la plupart des aspects du jeu soient terminés, certains ne le sont pas encore.

Durant l'année, il nous a été difficile de trouver un équilibre de travail et

d'avancer de manière commune, souvent à cause de la charge de travail hors projet. Nous avons donc souvent dû reporter les assignations d'un membre à un autre en fonction de leurs disponibilités. Cela a impacté négativement le développement du jeu : en effet, certaines personnes s'étant plus familiarisées avec certains aspects du jeu que d'autres, l'ajout de fonctionnalités en a été ralenti.

Par exemple, l'implémentation des items a été faite par une personne de notre groupe seule, et donc la personne en charge du lore n'était pas assez familiarisée avec l'implémentation des items pour en ajouter efficacement. Ce n'est que vers la fin du temps imparti que les membres de l'équipe ont su libérer du temps pour se pencher vers une compréhension plus profonde de l'interface de chacune des fonctionnalités et travailler de concert.

Nous avons appris de nos erreurs et réalisons maintenant que ce problème est ce qui a ralenti notre groupe dans l'ensemble, et donc prendrons garde à ce point lors de futurs projets.

5 Conclusion

IsoCraft Story est un sandbox et jeu d'aventure sur une planète alien. Il se déroule dans un monde de voxels infini et constitué de couches, en perspective isométrique. Une des mécaniques clés est la progression du joueur grâce aux interactions qu'il peut avoir avec le terrain et à la fabrication unique d'objets. Presque tous les points du cahier des charges originel ont été implémentés, mais l'implémentation d'une partie du contenu reste à faire.

Voici maintenant le ressenti global de chacun des membres de notre équipe :

Nicolas

La réalisation du jeu fut globalement plaisante, j'ai eu de la chance de faire partie d'une équipe comme celle de WarpZone Entertainment. Je culpabilise néanmoins et regrette tout de même de ne pas avoir fourni autant de travail que les autres membres du groupe. Il me reste un peu de considération sur ce que j'ai apporté même si cela reste minime. Ce projet me bénéficia, merci, vraiment.

L'avancement était tout sauf linéaire, la réalisation avait relativement bien avancé au cours de l'année puisqu'on avait commencé un peu en avance, mais les retards se sont accumulés et il a fallu accélérer sur les derniers mois - pour ne pas dire "le". Personnellement 70% de mon travail avait été fait sur le dernier

mois et 50% sur la dernière semaine.

Mais finalement on a réussi à finir ce qu'il fallait faire à temps avec un résultat satisfaisant.

Romain

J'ai vraiment apprécié travailler sur le projet IsoCraft Story, notamment l'imagination et la création du lore, qui était une partie passionnante pour moi. J'ai trouvé la création de modèles 3D et d'items un peu moins intéressante. Bien que j'aurais préféré travailler sur un projet moins ambitieux, où j'aurais pu être plus utile et à l'aise dans mes contributions, je suis quand même satisfait de ce dernier, car j'y ai rencontré des personnes vraiment sympathiques et sporadifiantes.

« Le cormoran n'est pas l'apanage de la vérité »

Raphaël

J'ai personnellement adoré travailler sur ce projet. Il m'a certes pris beaucoup de temps, mais je suis fier du résultat, que ce soit sur mes parties ou celles de mes camardes. J'ai appris énormément de choses à l'occasion de ce projet.

Léo

Pour ma part, je suis celui qui a le plus préféré travailler sur les aspects plus techniques du jeu. Dans un sens je sais que j'ai contribué car j'ai aidé à préparer des points clés du jeu, mais je me sens aussi responsable des ralentissements dans le développement du jeu, car les fonctionnalités que j'ai implémentées ont mis du temps à être utilisées globalement au sein du groupe.

Cependant, le projet est loin d'être terminé pour moi, je compte continuer à y contribuer pendant longtemps encore, jusqu'à arriver à un point où je suis satisfait du résultat. Et je suis très rarement satisfait de l'issue d'un de mes projets, même si je dois admettre que ce jeu a le potentiel de devenir bien plus complet que ce qu'il est aujourd'hui, ouvert à plusieurs mises à jour au cours du temps.

Cette aventure m'aura appris bien des choses, de l'utilisation de nouveaux outils de travail à l'apprentissage d'un nouveau langage de programmation. Mais surtout, cela m'aura appris que - il faut bien le dire - j'apprécie Unity encore moins que je ne croyais déjà le détester. Le mindset qu'il semble être nécessaire

d'avoir, de contentement dans une multitude d'outils non optimaux, trop faciles d'accès donc très difficiles à modifier de manière avancée pour satisfaire ses besoins, n'est clairement pas pour moi.

Je dois me rendre à l'évidence des faits : à part le moteur de rendu de Unity qui nous économise forcément beaucoup de temps, toutes les fonctionnalités du moteur que j'ai utilisées lors de ce projet m'ont fait perdre du temps par rapport à mon rythme de développement habituel, et j'ai presque toujours fini par les recoder, ce qui va à l'encontre du but initial d'un moteur fait pour être facile et rapide à utiliser.

C'est pourquoi je dis que, bien qu'allant continuer à travailler sur IsoCraft Story, je vais le faire avec comme but ultime de me passer de moteur de jeu.

6 Annexes

Lien repo GitHub : https://github.com/RaphoufouLeFou/Isocraft_Story

Lien site web : https://d-002.github.io/isocraft-story-website