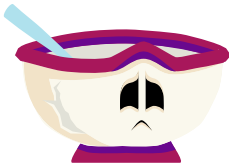
EPITA

RAPPORT DE PROJET

SeedWorld

SOUTENANCE FINALE
Par M.D.W.S.



Leyre Arnaut
Cruciani Léa

Marchetti Gauthier
Tripier Léo

À rendre pour le 17 juin 2021

Table des matières

1 Introduction

Vous avez pour projet de concevoir un jeu vidéo en 2D utilisant une vue de dessus ? Ou bien vous avez besoin d'un support pour vos jeux de rôle pour être davantage immergé dans votre partie ? SeedWorld est donc un logiciel qui pourrait vous intéresser. Son but est de générer une carte du monde de votre choix, vous aurez la possibilité de choisir le type de monde que vous souhaitez : féerique, médiéval, futuriste. Et il vous sera possible de manipuler les pourcentages des différentes régions, si vous souhaitez plus d'océans, de plaines, de zone montagneuses. Vous choisirez la dimension de votre monde en fonction des choix qui s'offrent à vous et le logiciel vous enverra sous forme d'une image JPG ou PNJ la carte finale. Sur cette image finale il vous sera possible de visualiser différentes infrastructures qui seront uniques en fonction du type de monde que vous avez sélectionné et qui seront typiques de chaque région.

1.1 Les membres de l'équipe

1.2 Léo Tripier

Lorsque les membres du groupe m'ont parlé de génération procédurale comme thème central de notre projet je ne savais pas encore de quoi il s'agissait. Je me suis donc documenté rapidement sur le sujet et j'ai vite vu que la génération procédurale était un thème passionnant et très complet, permettant de créer des choses très diversifiées. J'ai donc adhéré au projet *SeedWorld*.

1.3 Léa Cruciani

Passionnée de dessin et friande de jeux vidéo. La création des maps que j'explorais au fur et à mesure de mes parties m'a toujours intéressée. Quand j'ai découvert que la génération procédurale permettait de créer des maps très vastes, de manière aléatoire, ce projet de créer un logiciel qui utilise cette même méthode va être super intéressant à faire et j'ai hâte de m'y atteler.

1.4 Gauthier Marchetti

Fort de ma (trop ?) grande passion du jeu-vidéo, j'ai pu, à de nombreuses reprises côtoyer le principe de génération procédurale. Jusqu'à très récemment, ce terme semblait mystique et son principe obscur. Après des recherches préalables, le sujet s'est avéré passionnant et plein de défis. Défis que j'ai hâte de relever !

1.5 Arnaut Leyre

J'ai vu de très nombreuses vidéos de « jam » ou des développeurs utiliser des procédés de génération procédurale. Ces vidéo m'ont donné envie d'en apprendre plus sur le sujet et de m'y essayer.

1.6 Présentation du sujet

Notre choix de sujet s'est porté sur la création de contenu numérique à une grande échelle , de manière automatisée répondant à un ensemble de règles définies par des algorithmes. Pour ce faire nous avons choisis de créer des cartes de jeu utilisable pour des RPG en 2D vue du dessus ou bien des jeux de rôles.

1.7 Pourquoi ce sujet ?

Nous avons décidé de choisir ce sujet car nous trouvons le principe la génération procédurale intéressante. Elle peut avoir de multiple applications, notamment en infographie, où elle est utilisée pour créer des textures, ou dans la mode, pour créer des motifs imprimable sur des vêtements. La génération procédurale peut créer différentes structures, texture, univers, de manière unique. Elle peut être utilisée pour créer des végétaux, ou bien des personnages, comme dans le jeu Spore qui permet la rencontre de monstres uniques à chaque partie. L'aspect qui nous a le plus intéressé est surtout la possibilité de créer des mondes, comme dans Minecraft, où la carte est unique, vaste et riche, le tout, grâce à la génération procédurale. De plus les membres de notre groupe sont sujets à jouer à des RPG et de ce fait nous étions davantage intrigués par la génération des cartes du monde dans lequel nos personnages évoluent. C'est pour ces raisons que nous avons décidé de nous lancer dans le projet *SeedWorld* qui va générer des mondes uniques pour des jeux RPG ou encore des jeux de rôles comme Donjons & Dragons !

2 État de l'art

La génération procédurale est utilisée dans de très nombreux jeux est parmi les différentes de méthodes dénombrées, certaines sont plus populaires que d'autres dûent à leur simplicité ou à leur efficacité.

2.1 Les applications de la génération procédurale

La génération procédurale est principalement utilisée dans deux secteurs de l'industrie du divertissement : le cinéma et le jeu-vidéo. Pour le cinéma, elle est utilisée pour les effets spéciaux. La génération procédrale permet de créer ou de recréer des environnements fidèles à la réalité. En effet, pour donner une impression de réel, l'aléatoire maîtrisé permet de s'approcher au plus près de la réalité. Dans le jeu-vidéo, l'utilisation de la génération procédurale a d'autre applications. Elle est utilisée, comme au cinéma, pour créer des environnements virtuels, en 2D ou 3D, pour la création de carte, comme dans Rogue, Minecraft, ou Rust. Cependant, la génération procédurale est aussi utilisée dans les animations, elle est alors appelée animation procédurale, exemple dans Spore et le déplacement des personnages.

2.2 Bruit de Perlin

Le bruit de Perlin est un bruit dit de gradient. Il s'oppose au bruit de valeur. Ce procédé a été inventé par Ken Perlin en 1985 après avoir travaillé sur les effets spéciaux du film TRON. C'est une fonction de texture pseudo-aléatoire : elle permet d'introduire de la variabilité aléatoire dans un jeu de données tout en restant sous le contrôle de l'utilisateur. Le bruit de Perlin est notamment utilisé dans l'aléatoirisation de texture, comme les sprites de feu ou de fumée.
Principes de l'algorithme :

— Définition d'une matrice avec des vecteurs de gradient aléatoires : Il s'agit de créer une "grille" de taille n, qui, pour chaque "nœuds" de la grille, y associe un vecteur à deux dimensions. Ces vecteurs correspondent en faite aux vecteurs unitaires : (-1,0) ; (1,0) ; (0,-1) ; (0,1) choisis aléatoirement. Pour déterminer les vecteurs de façon aléatoire, il est utile d'utiliser un générateur de nombre aléatoire, ou RNG(random number generator).
— Calcul du produit scalaire entre le vecteur de gradient et le vecteur distance : Pour un point que nous appellerons p, il faut repérer sa position sur la grille. Par la suite, pour chaque nœud-sommet de la case identifiée, l'on doit calculer le vecteur distance entre le point p et le nœud-sommet. Enfin, il faut calculer le produit scalaire entre le vecteur de gradient au nœud et le vecteur de distance. Ici, nous allons faire quatre

calculs de vecteur distance ainsi que quatre produits scalaires. Nous sommes en effet dans le cas bidimentionnel, où chaque case est un carré fait de quatre nœuds-sommets.
— Interpolation de ces valeurs : L'interpolation sert à lisser les résultats. En effet, on souhaite obtenir des transitions douces entre les différentes zone de la carte. Pour cela, on utilise une fonction de lissage sur les données.

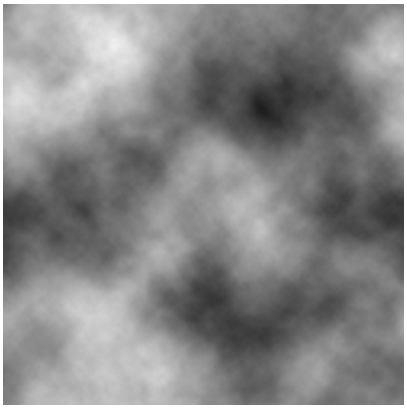


FIGURE 1 – Image en sortie du bruit de Perlin

2.3 Diagramme de Voronoï

Le diagramme de Voronoï (nommé d'après son créateur Gueorgui Feodossievitch Voronoï, un mathématicien russe de la seconde moitié du XIXe siècle). Le diagramme de Voronï permet un découpage particulier d'un espace. Ce découpage repose sur des "germes" des points situés dans l'espace qui serviront de base pour la création des différentes parcelles qui vont composer notre diagramme. Après avoir placé les "germes", on définit une région autour de ceux-ci incluant tout les points plus proche du germe sur lequel on se place. Cela permet de recouper un espace en différentes régions de différentes formes. Ce principe du diagramme de Voronoï pourrait nous être très utile pour générer des cartes toujours plus variées. De plus cela permettrait d'éviter que les cartes soient découpées par case carrée ou hexagonale comme souvent ce qui peut certes avoir un bon rendu mais qui donne trop souvent un découpage trop peu naturel. Pour construire un diagramme de Voronoï il existe plusieurs algorithmes. Celui qui nous semble le plus intéressant pour notre projet est appelé Algorithme de Fortune. L'idée principale de cet algorithme est de balayer le diagramme avec une droite (horizontale ou verticale). Chaque fois que la droite passeras sur un germe, on définira un arc de cercle. Lorsque deux arcs de cercle se croiseront, on utilisera leur croisements pour définir la droite separant les 2 régions. Lorsqu'un troisième arc de cercle se croisera en u point avec les deux précédents, on saura qu'il s'agit de la fin du segment.

2.4 L-System

Le L-System ou système de Lindenmayer, tient son nom du biologiste Lindenmayer qui utilisait des grammaires (un ensemble fini de symbole avec des règles) pour étudier le développement des végétaux, notamment des levures, des algues ou encore des champignons, avec une approche mathématique. Son utilisation a été ensuite déviée pour créer des végétaux notamment avec des fractales.

FIGURE 2 – Génération de végétaux en 3D à l'aide d'un L-System

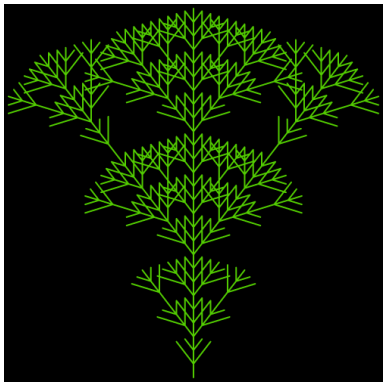


FIGURE 3 – Génération d'un arbre avec des fractales

Au final, seulement le bruit de Perlin a été utilisé pour la génération procédurale de la carte monde.

3 Notre Projet

3.1 Ce que nous allons réaliser

Notre projet a pour but de créer une carte générée de manière procédurale, sous forme d'une image. L'utilisateur, via une interface, peut choisir son mode de création : à partir d'une graine de génération (*seed*), selon des critères intelligibles (nombres de continents, proportion d'océan, proportion des différents biomes ...), ou encore de manière totalement aléatoire. L'utilisateur décidera de laisser le programme placer ou non des villes sur la carte, en fonction des conditions de l'environnement. De plus, toujours à la guise de l'utilisateur, le programme pourra créer les noms des villes. Pour cela, il utilisera un autre pan de la génération procédurale, son emploi sur le texte.

3.2 Prévisions de la répartition des taches

Soutenance 1	Léa	Arnaut	Léo	Gauthier
Génération des régions	X	30%	30%	X
Génération/Placement des villes/Sprites	30%	X	X	30%
Graphismes	30%	X	X	30%
Interface	X	30%	30%	X
Site web	50%	X	X	X

TABLE 1 – Répartition des tâches S1

Soutenance 2	Léa	Arnaut	Léo	Gauthier
Génération des régions	X	80%	80%	X
Génération/Placement des villes/Sprites	80%	X	X	80%
Graphismes	70%	X	X	70%
Interface	X	50%	50%	X
Site web	80%	X	X	X

TABLE 2 – Répartition des tâches S2

Soutenance finale	Léa	Arnaut	Léo	Gauthier
Génération des régions	X	100%	100%	X
Génération/Placement des villes/Sprites	100%	X	X	100%
Graphismes	100%	X	X	100%
Interface	X	100%	100%	X
Site web	100%	X	X	X

TABLE 3 – Répartition des tâches S3

3.3 Répartition finale des tâches

Soutenance 1	Léa	Arnaut	Léo	Gauthier
Génération des régions	X	30%	30%	X
Génération/Placement des villes/Sprites	30%	X	X	30%
Graphismes	30%	X	X	30%
Interface	X	30%	30%	X
Site web	50%	X	X	X

TABLE 4 – Répartition des tâches S1

Soutenance 2	Léa	Arnaut	Léo	Gauthier
Génération des régions	X	80%	80%	X
Génération/Placement des villes/Sprites	80%	X	X	80%
Graphismes	40%	X	X	40%
Interface	X	60%	60%	X
Site web	80%	X	X	X

TABLE 5 – Répartition des tâches S2

Soutenance finale	Léa	Arnaut	Léo	Gauthier
Génération des régions	X	100%	100%	X
Génération/Placement des villes/Sprites	100%	X	X	100%
Graphismes	100%	X	X	100%
Interface	X	100%	100%	X
Site web	100%	X	X	X

TABLE 6 – Répartition des tâches S3

4 Génération procédurale des biomes

4.1 Première soutenance

4.1.1 Arnaut Leyre

La génération de bruit similaire au bruit de Perlin a été parmi les priorités de cette soutenance.
Pour cela la première étape est de générer une matrice résultat nulle et une matrice *seed*. Cette dernière a plusieurs vocations. Tout d'abord elle assure une possible inclusion du concept de noyaux de génération pour la seconde soutenance. Mais elle sert aussi de base à notre prototype de bruit organique aléatoire.
À partir de cette matrice obtenue de manière aléatoire nous allons pouvoir générer notre résultat. Pour s'assurer de la cohérence et de la fluidité du résultat il faudra faire plusieurs passages de comparaison linéaire appelés *octave*. On va donc parcourir la matrice et pour chacun des points on va y appliquer plusieurs *octaves*. Pour chaque *octave* on va prendre des points-*échantillons* dont la distance est *la distance maximale* par rapport à notre point d'étude. C'est-à-dire un point situé à une distance équivalente à la taille du côté de la matrice. Si pour ce point la distance dépasse le bord de la matrice on fait le tour et on continue à compter de l'autre côté de la matrice. Cette étape est cruciale car elle est à l'origine d'une des propriétés les plus intéressante du bruit de Perlin. En effet si on regarde attentivement les bords de l'image on observe qu'ils correspondent parfaitement au bord opposé. Cette propriété permet une transition, ce qui est très utiles dans le cas d'un planisphère qui est une représentation en 2 dimensions d'une sphère.

FIGURE 4 – Image en sortie du bruit de Perlin

La distance maximale est divisée par chaque *octave*, permettant aux *échantillons* d'être de plus en plus proches et ainsi aux formes de se créer. Ceci implique aussi qu'il y a un nombre d'*octave* maximal qui peut être efficace, au-delà de ce nombre le code compare chaque pixel à lui-même. Ainsi pour les autres cas le programme compare le point d'étude aux différents *échantillons* et détermine une valeur moyenne. Le nombres d'*échantillons* correspond à la dimension de la matrice donc dans notre cas 2.
Les valeurs moyennes sont ensuite multipliées par une *échelle* et ajoutées les unes aux autres pour enfin être sommées au résultat. L'*échelle* doit être divisée par un *biais* pour que les détails soient de plus en plus fin. Le *biais* et le nombre d'*octave* sont également les clés qui permettent de contrôler l'aspect de notre bruit de Perlin. Plus on a d'*octaves*, plus on a de détails, plus le *biais* est petit, plus le terrain généré est rugueux et aléatoire.

4.2 Deuxième soutenance

4.2.1 Léo Tripier

La nouveauté implémentée pour cette soutenance : les biomes ! Léo s'est attelé à la création d'un prototype de génération de biomes. Se basant sur une suggestion d'Arnaut, il a utilisé une seconde carte générée aléatoirement (appelée carte des températures) en utilisant les même paramètres de biais et d'octave que pour la carte précédente mais en utilisant une seed aléatoire différente.
Ensuite la fonction *gen_ biomes* sera appelée en passant en paramètre les deux matrices. L'une pour les reliefs (océan,plaine, montagne) l'autre pour les "températures", des *float*

entre 0 et 1. La fonction va donc parcourir ces deux matrices simultanément et croiser les informations contenues afin de générer une nouvelle matrice de *int* (représenter dans le code par un pointeur *int**). Cette matrice est donc remplie de *int* entre 1 et 9 (compris) qui représente donc le type de biome.

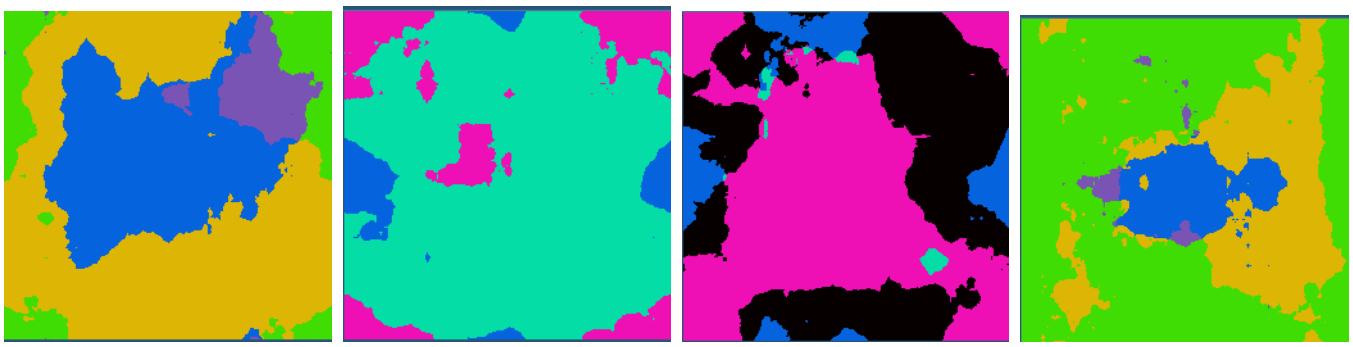


FIGURE 5 – Différentes cartes avec des biomes

Pour l'instant rien n'est fixé, les types de biomes ne sont que des entiers et ne sont pas encore associés à quoique ce soit mais cela sera ajouté pour la soutenance finale bien évidemment. Pour l'instant le code associe simplement une couleur ne correspondant à rien de particulier cela permet juste de vérifier que l'algorithme fonctionne correctement, ce qui est le cas, on observe bien des zones distinctes comme attendu.

4.2.2 Arnaut Leyre

Lors de cette soutenance la génération et la manipulation des seeds ont été ajoutées. Pour cela la seed est décidée selon la date puis grâce à la librairie *rand* les futures valeurs seront décidées en fonction de la seed et de l'ordre de leur génération. Cette solution est particulièrement efficace car tous les codes de la soutenances précédente dépendent de cette librairie.
De plus le problème d'affichage des couleurs a été réglé. Il s'agissait d'un problème dût au format de l'image utilisé pour enregistrer le résultat et donc la palette de couleur était limitée.

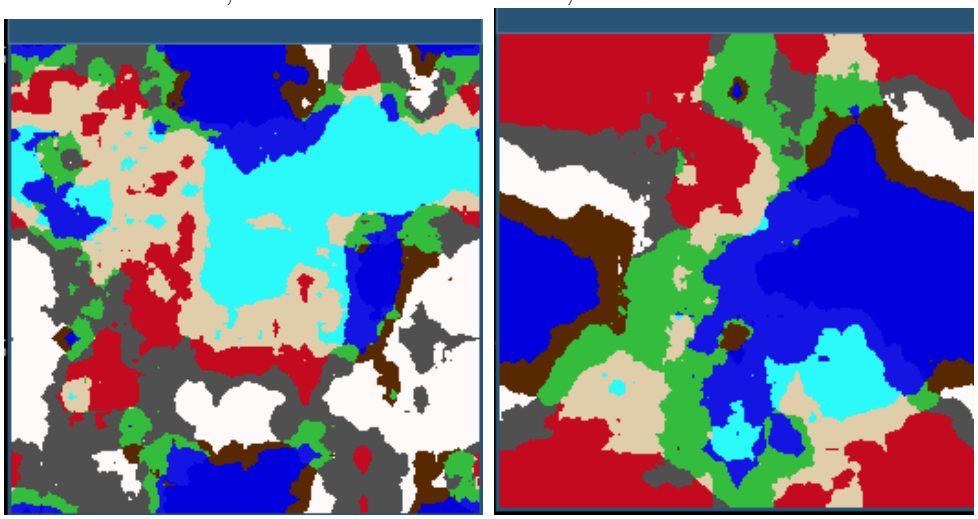
4.3 Soutenance finale

4.3.1 Léo Tripier

Pour cette soutenance, Léo a créé une fonction permettant de contrôler la proportion d'océan, de terre et de montagne ainsi que la proportion de températures froides, modérées ou chaudes. La signature de la fonction est la suivante : *float* seuil(size_t height, size_t width,float* mat, int ocean, int plains, int mountains)* . Le pointeur *float** renvoyé pointe sur

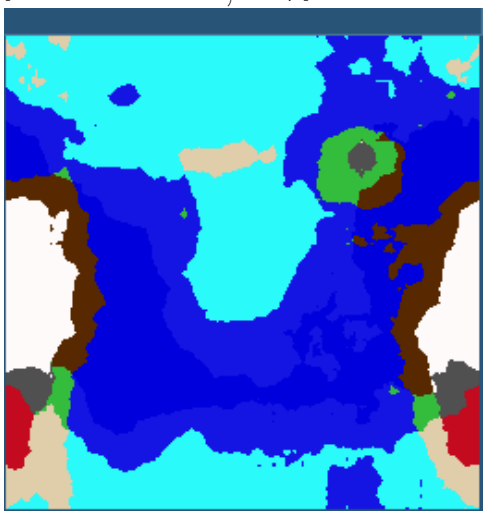
un tableau de deux valeurs, la première valeur sera le seuil fixé pour atteindre la proportion souhaitée d'océan, la deuxième valeur sera le seuil fixé pour atteindre la proportion souhaitée de terre constructible. La proportion de montagne sera naturellement celle souhaitée car chaque case n'étant ni de l'océan ni de la terre constructible est forcément une case de montagne. Cette fonction s'applique aussi pour les proportions de températures car elles reposent sur les mêmes structures et algorithmes.
Le principe de cette fonction est le suivant : à partir de la matrice renvoyée par l'algorithme de bruit de Perlin d'Arnaut, on établit un histogramme des valeurs. On ne normalise pas cet histogramme car avec l'approximation due à l'imprécision des nombre flottant, cela ferait disparaître certaines valeurs. Ensuite, on parcourt l'histogramme en sommant les valeurs et pour chaque itération on calcule la proportion de pixels comptabilisée jusque là en divisant la somme totale par le nombre de pixels. Dès que l'on est sure qu'il y a au moins la proportion souhaitée de pixels en dessous de ce seuil (valeur du pixel) on le fixe comme étant le seuil d'océan, on continue ensuite l'itération jusqu'à trouver le seuil pour obtenir la bonne proportion de terre constructible par le même principe. On arrête ici le parcours de l'histogramme puisque la dernière proportion n'a pas besoin de seuil. On retourne ensuite le pointeur qui pointe vers le premier seuil.
Ensuite, lors du parcours de la matrice du bruit de Perlin, on comparera chaque valeur de la matrice à ces seuils pour déterminer si le pixel en question est un pixel d'océan, de terre ou de montagne. Tout ce processus est effectué une deuxième fois avec une matrice de Perlin différente et des proportions différentes pour les températures. À la fin, on obtient bien une image avec les proportions souhaitées pour les oceans/terres/montagnes ainsi que pour les températures. Voici les résultats obtenus :

33% océans ; *33%* terres ; *33%* montagnes
33% climat froid ; *33%* climat modéré ; *33%* climat chaud

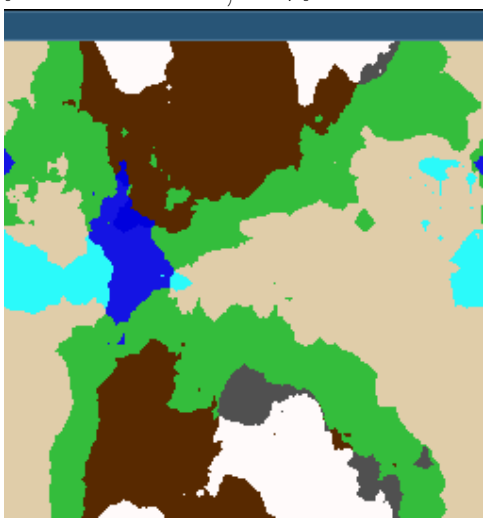


80% océans ; *10%* terres ; *10%* montagnes

33% climat froid ; *33%* climat modéré ; *33%* climat chaud



10% océans ; *80%* terres ; *10%* montagnes
33% climat froid ; *33%* climat modéré ; *33%* climat chaud



Voici le code hexadécimal des couleurs ainsi que leur correspondance :

- *#0000DD* océan froid
- *#1414E3* océan
- *#2BFAFA* océan chaud
- *#582900* terre froide
- *#34BD3C* terre
- *#E0CDA9* désert
- *#FFFAFA* montagnes gelées
- *#505050* montagnes
- *#C30A1E* volcan

Comme on le voit sur ces images, les proportions sont assez bien respectées, lorsque l'on demande 80% d'océan on observe bien une très large prédominance des océans. Idem lorsque

l'on demande 80% de terres on observe une large prédominance des biomes de terres (terres froides, terre et désert) et cela en conservant bien des proportions équivalentes (à peu près) pour les températures (environ 33%). Les proportions ne sont pas d'une précision parfaite mais les imprécisions étant légères, visuellement le résultat est celui escompté. L'origine de ces imprécisions vient du calcul de proportion. Prenons un exemple, on cherche à obtenir 30% d'océan, lors du parcours de l'histogramme, on calcul que 29% des pixels ont une valeur inférieur ou égal à 0.30 par exemple, à la prochaine itération on comptabilise les pixels égaux à 0.31 en plus et la le pourcentage passe à 32% car il y à beaucoup de pixels égaux à 0.31. Ainsi la fonction fixera le seuil à 0.31 mais le pourcentage sera légèrement supérieur à celui demandé.
L'une des solutions pour résoudre ce problème d'imprécision serait d'augmenter la précision au millième près. Cependant, la répartition des valeurs faites par l'algorithme de Perlin d'Arnaut fait que cette imprécision n'est jamais très grande. Il n'est donc pas nécessaire d'alourdir l'histogramme, cela prendrait plus de place en mémoire et augmenterait le nombre d'itération pour parcourir l'histogramme. Cela n'étant pas réellement nécessaire pour le bon fonctionnement de notre programme Léo à fait le choix de ne pas augmenter le degré de précision pour conserver une meilleure efficacité.
Une autre source d'imprécision est due à la conversion de type flottant vers des entiers et inversement. Ces conversions sont nécessaires pour pouvoir construire l'histogramme, nécessité d'avoir des entiers comme index de l'histogramme (première conversion), nécessité que les seuls soient des nombres flottant puisqu'ils seront comparés à des flottants (deuxième conversion). La principale difficulté que Léo a rencontré à été due à ces conversions et à la manière dont on calcule la proportion . Dans certains cas, la conversion effaçait la donnée (mise à zéro), ce qui donnait des maps uniformes. Le second problème, venait du calcul de la proportion. Au début, Léo voulu normaliser l'histogramme, le problème était que certaines valeurs de l'histogramme étaient mises à zéro lors de la normalisation car trop petites relativement au nombre de pixel total. Il a donc été nécessaire de calculer la proportion à chaque itération, la proportion calculée pour une itération i est le nombre de pixels ayant une valeur inférieur ou égal à i divisée par le nombre total de pixels.

4.3.2 Arnaut Leyre

5 Génération procédurale des villes

5.1 Première soutenance

5.1.1 Léa Cruciani

Pour la génération de ville avec des réprésentations aléatoires et uniques, Léa a pris en charge la création des routes. Pour ce faire quatre fonctions ont été créées :

- *nb_rand(int min, int max)* qui donne un chiffre dans l'intervalle min et max inclus.
- *define_dir(int x, int y, int cols, int rows, int vertical)* qui donne la direction de la route qui va être créée (si celle-ci sera vers le haut, le bas, à droite ou bien à gauche).
- *draw_road(int* map, int x, int y, int r_size, int cols, int rows, int vertical, int direction)* qui va permettre, comme son nom l'indique, de construire une route.
- *build_roads(int* map, int rows, int cols)* qui va se charger de lier les routes et de construire tout le système routier de la future ville.

Le principe de cet algorithme est de construire un réseau routier unique qui ensuite pourra permettre de placer des maisons. La fonction principale *build_roads* va prendre un tableau de 0 (*int* map*) à une dimension qui sera considéré comme la zone dans laquelle la ville devra être bâtie. L'implémentation *row-major order* qui offre la possibilité de traiter un tableau à une dimension en une matrice sera utilisée tout au long de ce projet. Pour que cela soit possible il est nécessaire que *build_roads* prenne en paramètre le nombre de colonnes (*int cols*) et le nombre lignes (*int rows*) qui vont être utilisés par notre future matrice.

La fonction *build_roads* va ensuite créer dans un premier temps la route principale : la création des variables *int x* et *int y* représenteront le début de notre première route, et *int len* sera sa longueur. Pour permettre aux routes d'avoir des dispositions uniques, ces valeurs seront générées de manière aléatoire à l'aide de la fonction *nb_rand* :

- x et y devront être choisies entre 0 et le nombre de lignes(ou colonnes), divisé par deux - un, car on veut exclure le fait que x(ou y) peut être égal aux nombres de lignes(ou colonnes).
- *len* sera quant à elle choisie entre la moitié de la longueur totale de notre tableau (*cols*rows/2*) et sa longueur totale (*cols*rows*) ;

Une fois les coordonnées de départ et la longueur de la route principale obtenues il faut désormais indiquer sa future orientation qui sera générée de manière aléatoire par *int vertical= rand()%2*, si *vertical = 1* alors la route sera verticale sinon elle sera horizontale. Cela est maintenant au tour de sa direction qui va lui être définie avec l'aide de la fonction *define_dir* qui va prendre en paramètre les coordonnées x et y de la route, le nombre de co-

lonnes *cols*, de lignes *rows* de notre matrice et *vertical* pour savoir l'orientation de la route. Si elle est verticale, seulement notre coordonnée x sera modifiée. L'utilisation de la variable définie *MIN_LENGTH* qui sera égale au maximum entre la hauteur et la longueur d'une maison (dans notre cas 48 pixels donc 48), sera nécessaire pour privilégier les directions des futures routes. En effet si notre route est verticale et commence aux coordonnées x *=1* et y*=0* cela ne sera pas productif qu'elle se dirige vers le haut (car elle fera 1 de long) et la deuxième route que l'on va contruire n'aura pas l'embaras du choix pour ses coordonnées de départ. Ainsi la direction de la route s'opérera pour priviligier une longeur importante.

Par exemple : notre route est orientée de manière verticale, c'est la coordonée x qui va varier. Si x+ *MIN_LENGTH* et x-*MIN_LENGTH* sont toujours définies dans la matrice alors la route peut tout à fait aller en haut ou en bas : notre direction sera générée de manière aléatoire ! *final_dir= rand()%2* si x-*MIN_LENGTH* est hors de la matrice cela signifie que notre route est trop proche du bord haut de la matrice ! Sa direction sera donc vers le bas *final_dir=0* sinon elle sera dirigée vers le haut. Il en va de même si notre route est orientée de manière horizontale sauf que cette fois-ci c'est la coordonnée y qui va varier.

Désormais en possession de la longueur de la route, ses coordonnées de départ, sa direction, et son orientation, il est possible de la dessiner. C'est à ce moment que la fonction *draw_road* participe à la tâche. Elle va prendre en paramètre le nombre de colonne et de ligne de notre matrice, les coordonnées du point de départ de notre route, son orientation, sa direction et la longeur de la route *r_size*. Si la route est verticale (horizontale) et est dirigée vers le(la) haut(gauche) $x(y)$ sera décrémentée ,dans le cas contraire incrémentée. À chaque itérations les coordonnées x et y seront testées pour savoir si elles sont toujours définies dans la matrice, ainsi si la longeur de la route désirée est trop grande la boucle sera automatiquement stoppée. Cette fonction renvoie la coordonée de la fin de la route (x si verticale, y si horizontale).
La route principale est tracée ! Il faut attaquer les routes subsidiaires qui auront pour point de départ des coordonnées entre le début et la fin de la route qui a précédé. Léa a décidé de limiter le nombre de route créer à un nombre choisi arbitrairement pour le moment, il sera modifié en fonction des besoins des prochaines soutenances et des modifications. Tant que le nombre total de route(*int nb_roads*) sera strictement supérieur à 0 il faudra continuer à en créer.

L'orientation de la route(r2) qu'on va créer va être le contraire de celle qui l'a précédée(r1). Si r1 est horizontale : r2 sera verticale. Les points de coordonnées de r2 devront obligatoirement appartenir à la route r1 ainsi il n'y aura pas de route isolée(ce qui fait sens). Si r1 est horizontale seule la coordonée y sera modifiée pour le point de départ de r2. Si r1 est dirigée

vers la droite et est assez longue pour que $y+MIN\_LENGTH$ alors la coordonnée y sera choisie de manière aléatoire avec nb_rand entre $y+MIN\_LENGTH$ et *end* qui symbolise la fin de r1. Dans le cas contraire y sera choisie de la même manière entre y et *end*. Il s'agira du même processus si r1 est verticale mais cette fois-ci x sera modifiée en conséquent. *define_dir* s'occupe de déterminer la direction de r2, l'orientation passée en paramètre de cette fonction devra être opposée à celle de r1 car la future orientation de r2 sera l'opposé de l'orientation actuelle de r1. Les coordonnées de départ de r2 sont récupérées, sa direction, son orientation sont définies également, il est possible de la tracer, avec bien entendu *draw_road* le retour de cette fonction sera récupéré par *end* qui sera la fin de r2. Une fois r2 tracée il faut actualiser *vertical* qui indiquera l'orientation de la prochaine route(r3) qui aura une orientation opposée à r2 : *vertical*= *!vertical* et enfin *nb_roads* sera décrémenté. Exemple d'exécution : matrice carrée de 50 par 50, nombre total de routes subsidiaires 20, les 6 indiquent le point de départ d'une route.

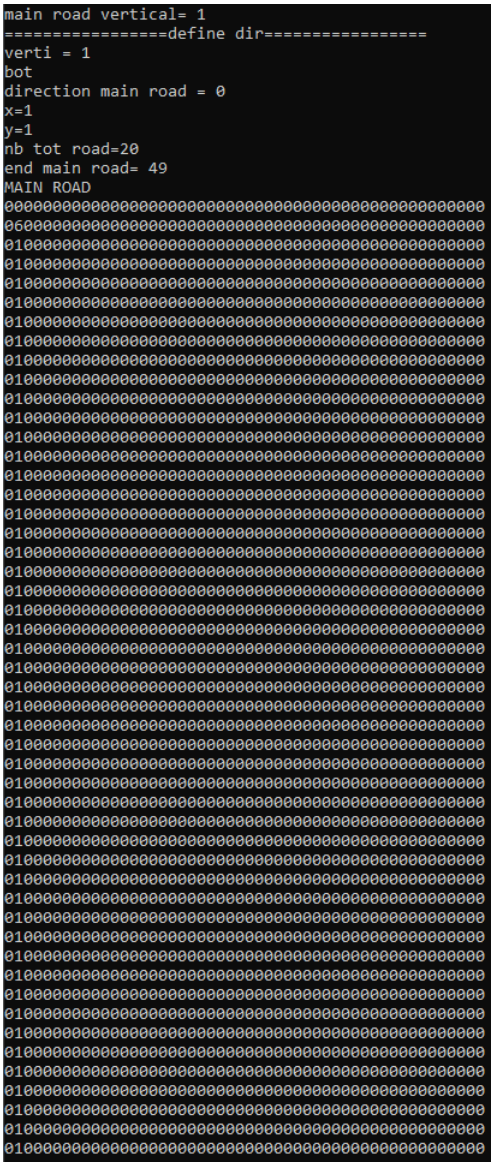


FIGURE 6 – Route principale créée

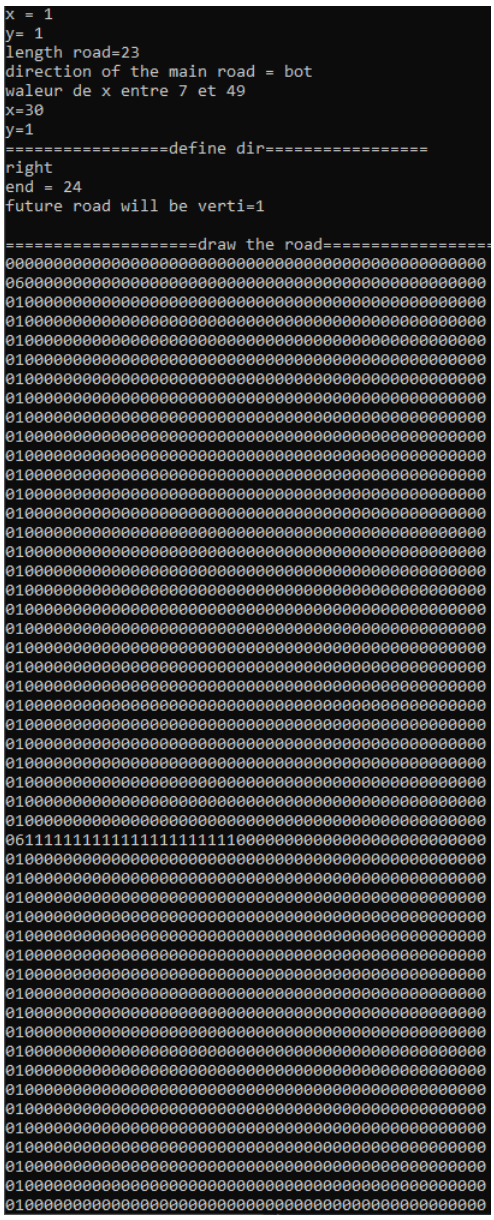


FIGURE 7 – Route subsidiaire créée

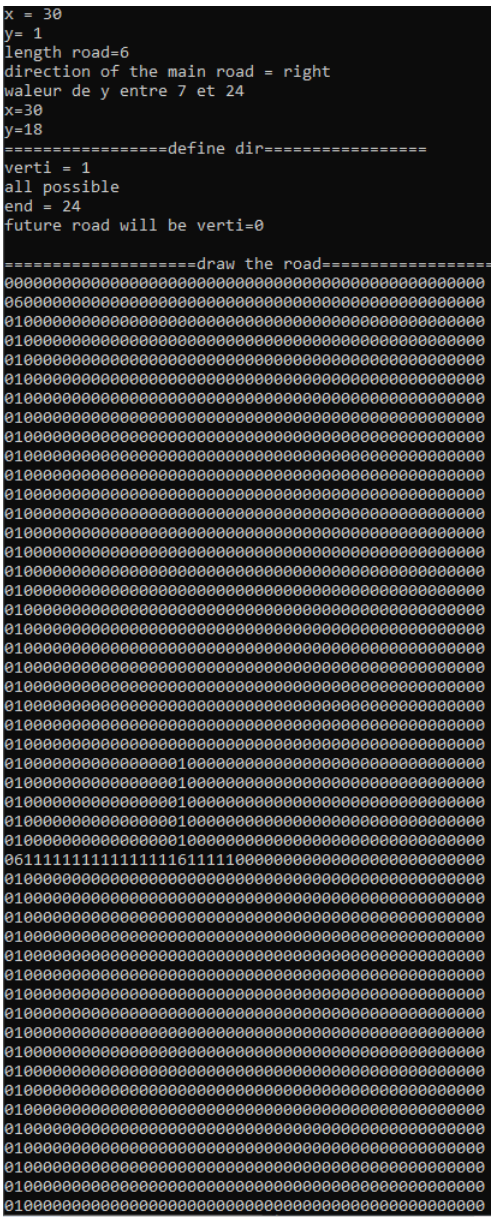


FIGURE 8 – Route subsidiaire créée

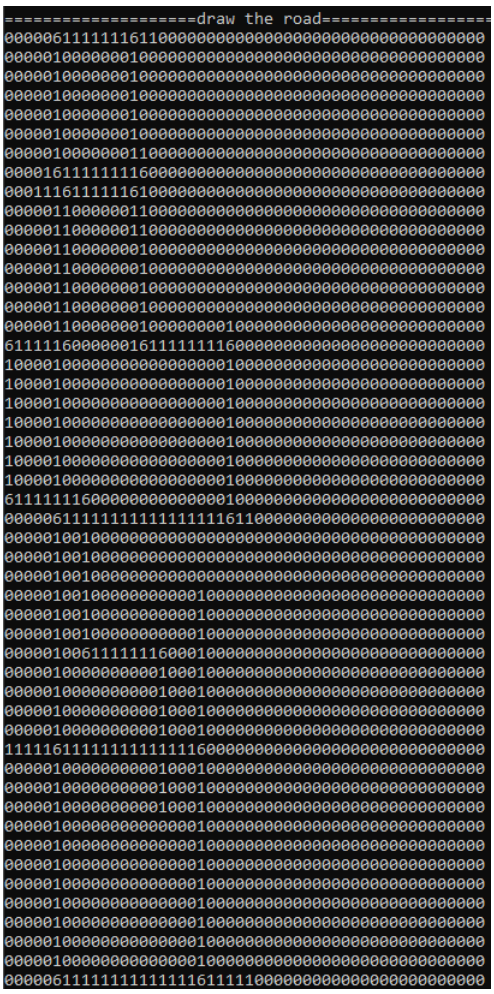


FIGURE 9 – Fin du processus

5.1.2 Gauthier Marchetti

Une fois la carte des routes de la ville créée, il faut placer les maison là où l'espace est suffisant. Pour ce faire, Gauthier a récupéré la matrice établie par Léa. Il s'est attelé à scanner les zones autour des intersection produites par les differentes routes. La fonction *find_inter* parcoure l'entièreté de la matrice à la recherche du caractère correspondant à une intersection.
Une fois l'emplacement de l'intersection déterminé, la fonction *draw_house* est appelée. Cette fonction est le cœur de la procédure de placement des maison. Elle se sépare en quatre sous-parties, correspondantes au quatre coins de l'intersection (supérieur gauche, supérieur droit, inférieur gauche, inférieur droit). Pour chaque sous-zone, la fonction scanne, selon la taille prédéfinie des maison, la surface nécéssaire au placement d'une maison. Si la surface est non conforme ou trop petite : traversée par une route, en bord de matrice, déjà occupé par une autre maison ; la procédure recommence avec une maison de taille inférieure, et cela jusqu'à ce que toute les tailles de maison ai été testées.
Pour ce faire, on intitialise deux paramètres *width* et *length* avec la largeur et la longueur de la plus grande maison, ainsi qu'un bouléen *valid* qui servira à vérifier si la surface traitée est conforme. Pour éviter des tours de boucle inutile, la surface minimale requise pour la plus petite des maisons est testée. C'est à dire que sans même regarder si la zone est obstruée on vérifie que l'espace brut est suffisant. Par la suite, on vérifie si la zone choisie n'est pas déjà occupé par une autre route ou une autre maison. Si c'est le cas, *valid* devient faux et arrête immédiatement les itérations inutiles, la procédure recommence avec la taille de maison suivante. Si les itération arrivent à leur terme et que *valid* est toujours vrai, alors s'enclenche le placement en soi des maisons, de manière matricielle.
Pour le placement des maisons de manière matricelle, nous avons établit notre propre convention : un caractère spécial, ici *4*, *5* ou *6*, sera placé dans le coin supérieur gauche de la surface alloué à la maison.

5.2 Deuxième Soutenance

5.2.1 Léa Cruciani

Pour cette deuxième soutenance Léa acorrigé les aglutinements des routes qui pouvaient poser problème pour l'insertion des maisons et aussi pour l'aspect réaliste du réseau routier. Pour ce faire différentes fonctions ont été ajoutées :

— *int not_on(int* map, int x, int cols, int vertical, int direction*
Cette fonction permet de vérifier qu'il n'y a pas une autre route autour du point de départ de notre future route subsidiaire, elle prend en paramètre la matrice d'entier représentant la carte de notre réseau routier, les coordonnées x et y de notre point de départ et enfin l'orientation de notre future route(horizontale ou verticale) et sa direction (droite/gauche, ou haut/bas). Si une case de notre matrice autour de notre point de départ vaut 1 alors on ne pourra pas construire une nouvelle route sans en empiéter sur une autre. *not_on* renverra donc 0 et 1 dans le cas contraire.

— *int is_constructible(int*map,int x, int y, int len, int cols, int rows, int vertical, int direction)*
Il s'agit de la fonction principale qui va parcourir la carte de notre réseau routier sur la longueur de notre future route pour estimer si celle-ci est assez espacée des autres pour pouvoir être construite. Elle prend en paramètre les coordonnées du point de départ de notre route, le nombre de colonnes et de lignes de notre matrice, l'orientation et la direction de notre route en construction. Dans le cas où notre route est verticale seulement x va être décrémentée(incrémentée) si la direction est en haut(bas). Dans le second cas notre route sera horizontale et y sera décrémentée(incrémentée) si la direction est vers la gauche(droite). Sur chaque point appartenant à la future route *is_constructible* fera appel à *constructible* qui va balayer les coordonnées de part et d'autre de celle-ci. À la fin du processus *is_constructible* renverra un 1 si la construction de la route est possible, un 0 dans le cas contraire.

— *int constructible(int*map, int x, int y, int fixed, int cols, int len, int i, int j, int max_pos, int vertical)*
Comme indiquer précédemment *construtible* va balayer les coordonnées autour de x et y sur une longueur qui est égale à la largeur minimale d'une maison, comme cela la possibilité d'insérer une maison sera plus élevée. Elle prend comme paramètre *int fixed* qui représente la coordonnée fixée sur la laquelle on va tester qu'il n'y ait pas de possibles routes déjà existantes de part et d'autre. Cette coordonnée fixée est importante car en fonction de l'orientation de la route celle-ci ne sera pas la même, dans le cas où elle est horizontale c'est le y qui sera fixée dans le cas contraire le x. Léa

a décidé de gérer les possibles intersections des routes. En effet, la fonction renvoie 0 si elle rencontre un 1(représentant une autre route) et cela exclut la possibilité de créer des intersections. Pour palier à ce problème Léa a inclus une condition qui stipule que si la longueur de la route qu'on est entrain de tester est supérieure à la largeur minimum d'une maison et que la coordonnée sur laquelle on est, vaut 1 et que celles de part et d'autre valent également 1 alors cela signifie que la route qu'on souhaite construire en croise une autre et sa construction est donc possible.

Les routes peuvent s'intersectionner sans créer des aglutinements. Cependant un problème reste à régler il est possible que certains réseaux routiers ne puissent pas être développés complètement dût à la génération aléatoire du point de cépart et de la longeur aléatoire d'une route. Cela peut entraîner des villes qui ne seront pas intéressantes à construire.

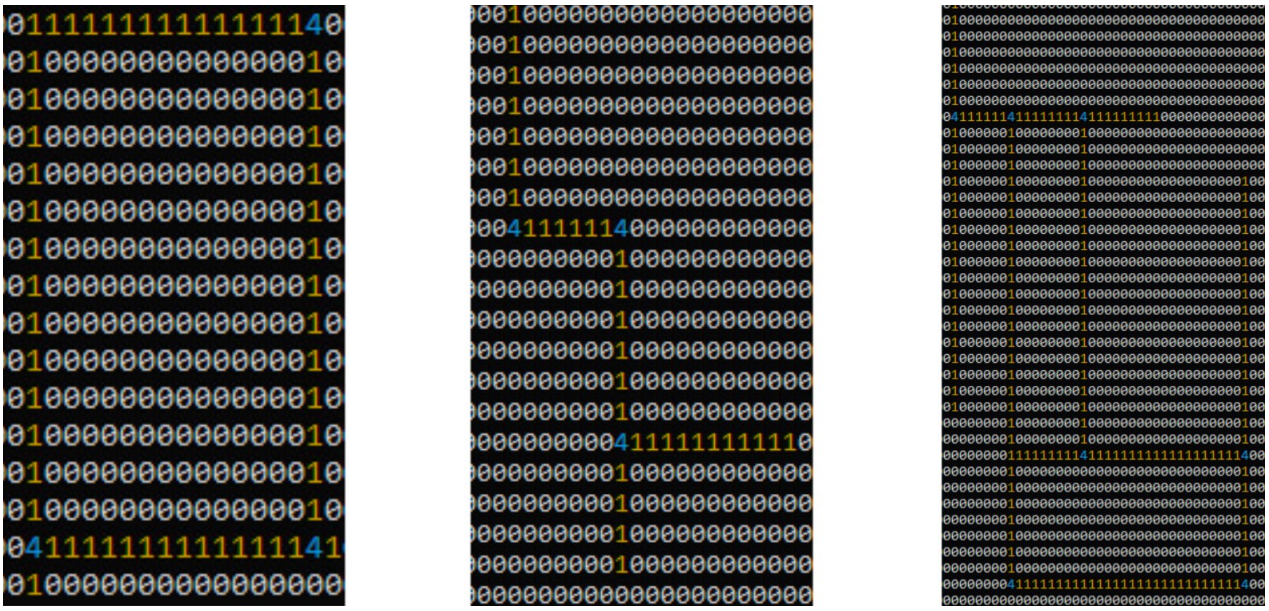


FIGURE 10 – Réseau routier peu développé et développé

D'autres fonctions ont été ajoutées pour permettre une meilleure lisibilité du code, ou pour permettre une meilleure approche de certaine partie du code.

— *define_len* permet de générer la longueur d'une route de manière aléatoire entre la longueur minimale d'une maison et sa longueur maximale qu'elle peut atteindre en partant de ses coordonnées de départ dans la matrice et non plus d'une longueur maximale définie manuellement.

— *define_begin* permet de générer aléatoirement le point de départ d'une route secondaire sur la principale.

Pour cette deuxième soutenance Léa s'est aussi chargée d'implémenter la recherche et la confirmation d'un emplacement constructible pouvant accueillir un ville. Pour ce faire il a été décider de fixer le nombre de villes qu'il est souhaité de créer, le nombre de villes différentes et enfin leurs tailles (nombre de colonnes et de lignes). Pour ce faire deux fonctions ont été créées :

— *int* find_fields(int* map, int cols, int rows)*
Cette fonction va parcourir la carte monde, si plusieurs villes sont constructibles sur un même x et y alors le choix de la ville se fera de manière aléatoire puis elle s'occupera de gérer la prochaine coordonnées sur les colonnes en ajoutant 2 fois la largeur de la ville qui a été créée sur y et le prochain x se verra ajouter 2 fois la hauteur de la ville qui a été construite. De cette manière les villes seront espacées et auront une zone d'influence ce qui empêchera les regroupements qui ne sont pas réalistes. Une fois le parcours de la carte terminée ,soit car le nombre de ville qu'on souhaite construire a été atteint soit car on arrive à la fin de la carte, la fonction renvoie un tableau de données qui contient les coordonnées x et y leurs nombres de colonnes et leurs nombre de lignes. Par exemple, si le nombre de villes que l'on souhaite construire est de 2 et que 2 villes ont été construites la tableau de sortie sera égal à

$$x1, y1, cols1, rows1, x2, y2, cols2, rows2$$

. Si seulement une ville a été construite :

$$x1, y1, cols1, rows1, -1, -1, -1, -1$$

, si aucune ville n'a pu être construite :

$$-1, -1, -1, -1, -1, -1, -1, -1$$

.

— *int buildable(int* map, int x, int y, int cols, int rows, int width, int height)*
Cette sous fonction est utilisée par *find_fields*, elle permet de tester à partir des

coordonnées et la taille de la future ville transmises par *find_fields* si celle-ci peut être construite. Elle parcourt la zone que devrait occuper la ville et si un seul point ne vaut pas 1 alors la zone n'est pas constructible et 0 sera renvoyé, dans le cas contraire 1 sera renvoyé.

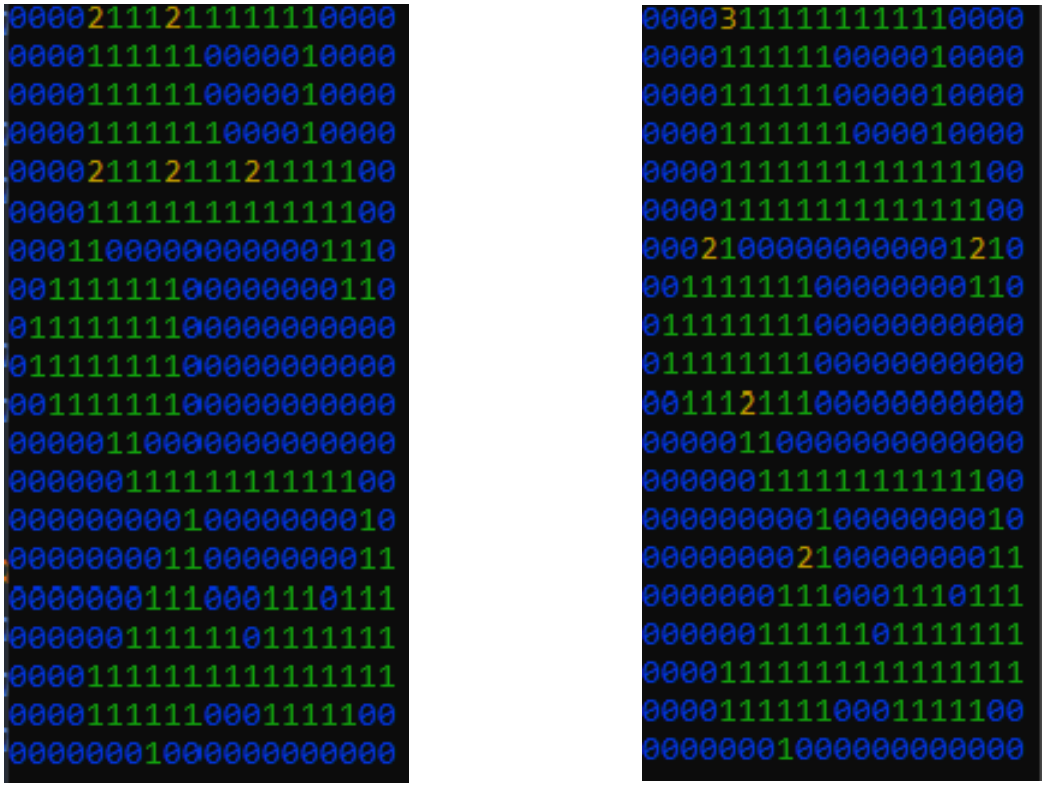


FIGURE 11 – Placement des villes 2=2*2 3=5*3 sur la même carte

5.2.2 Gauthier Marchetti

Lors de la soutenance précédente, une fois la matrice des routes créée, une procédure était appliquée qui parcourait la matrice des route afin de déterminer les zones disponibles à l'ajout de maison. Un phénomène d'agglutinement des maisons était observé. Afin de corriger ce problème, Gauthier a modifié le code de la première soutenance afin d'inclure une "zone morte" qui représente la place qu'occupe la maison sur la matrice. Ainsi, lors des futures passages de detection, les "zones mortes" ne pourront pas accueillir de nouvelles maisons.

L'échéance principale de cette soutenance a été la transcription d'image matricielle vers une image de type *BMP*. Pour cela, on parcourt la carte matricielle et en fonction de la valeur matricielle on affecte une certaine couleur au pixel : gris si c'est une route (*1* ou *4*), rose si c'est une case vide (*0*). La fonction *createSpriteRGB* initialise une nouvelle surface SDL de la dimension de la matrice *map*. Lors d'un premier parcours de matrice, les cases "basiques", qui correspondent aux routes et aux cases vides. Dans un second parcours de la matrice, on cherche les valeurs correspondantes aux maisons et la fonction de placement de sprite est appelée sur une image.

Il a été choisi de mettre un pixel rose pour les cases vides car le format Windows BMP ne supporte pas la transparance dans une image.

Le second objectif de cette soutenance a été le collage d'un sprite sur une image. Pour cela, on parcourt simultanément le sprite à coller ainsi que l'image receveuse. On transcrit alors pixel par pixel les valeurs de pixel du sprite sur l'image. La fonction *place_house* récupère l'image receveuse *img* ainsi que le token correspondant au type de maison à placé. Dans la surface SDL *house* est chargé le BMP correspondant au token de type de maison. En fin de procédure on obtient l'image correspondant au collage du sprite sur l'image receveuse.

5.3 Soutenance finale

5.3.1 Léa Cruciani

Pour cette soutenance, Léa s'est majoritairement occupée de la génération du nom des villes. Dans un premire temps il avait été dit que cela serait au travers d'un réseau de neuronnes que les noms des villes seraient générés pour qu'ils gardent une certaine cohérence dans l'ordre des lettres afin d'éviter un nom de ville qui pourrait ressembler à *rhbxz*. Afin

de garder une forme de logique quand à l'emplacement des lettres les unes par rapport aux autres en fonction du type de monde ,qui a été sélectionné par l'utilisateur, Léa et Gauthier ont fixé quelques règles basiques que sont les suivantes :

- chaque monde possèdera un tableau de consonnes avec toutes les consonnes que ce monde peut contenir(il en va de même pour les voyelles).

- si deux lettres du même type se suivent alors la troisième lettre sera d'un type différent(*ex : deux voyelles la troisème lettre sera une consonne*).

- si une consonne est choisie alors la prochaine lettre sera une voyelle.

- si la lettre précédente est un m, un l, un s, un f, un r il est possible qu'elle soit doublée si elle ne se trouve pas au début du mot.

- si notre lettre précédente est un q les deux lettres suivantes seront un u puis une voyelle.

- un y ne peut en suivre un autre(de même pour le u).

Une fois ces quelques règles fixées il faut maintenant coder la fonction *name_gen(int type)* qui permet de générer un nom d'une ville en fonction de son type de monde *type=1* :monde féérique,*type=2* :monde médiéval,*type=3* :monde futuriste. Pour mener à bien sa génération de nom cette fonction principale a besoin de différentes sous-fonctions :

- *char* finding_name(int p_voy,int type,int len,int* word)* :va appliquer toutes les règles vue précédemment. Elle va se charger de passer à la sous-fonction *choose_letter* le tableau de voyelles ou de consonnes dans laquelle la prochaine lettre sera générée aléatoirement. Pour savoir si deux lettres d'un même type se suivent les *d_v=1* si deux voyelles se suivent (0 dans le cas contraire) ou *d_c=1* si deux consonnes se suivent. Les lettres sont choisies tant que notre mot n'a pas atteint sa longueur fixée. S'il y a aucune obligation sur le choix d'une lettre celle-ci sera sélectionnée en fonction d'un pourcentage généré de manière aléatoire. Si celui-ci est au-dessus du pourcentage de voyelle alors la prochaine lettre sélectionnée sera une consonne(une voyelle dans le cas contraire). Les pourcentages minimums requis pour être une voyelle sont imposés par la fonction principale *name_gen()* qui en fonction du type de monde aura des pourcentages différents.

— *char* defined_tab_v(int type,*max_v)* :va générer le tableau de voyelles propre à chaque type de monde.

— *char* defined_tab_c(int type,*max_c)* :va générer le tableau de consonnes propre à chaque type de monde.

— *void choose_letter(char* tab, char* word, int max,int i)* :va choisir une lettre aléatoire entre 0 et *max* dans le *tab* correspondant et va la placer à l'index *i* du mot *word*.

5.3.2 Gauthier Marchetti

Le travail sur cette soutenance finale s'est décomposé en trois grandes parties : Les corrections, améliorations, optimisations sur les parties précédentes ainsi que la création des dernières fonctions nécessaires, ainsi que la mise en commun du travail des deux sous-groupes.

Pour les corrections, améliorations et optimisations, les fonctions concernées sont les suivantes :

draw_house2 : La fonction finale et la première n'ont plus grand chose en commun. En effet, la première version de cette fonction n'était pas totalement fonctionelle, peu optimisée et illisible. Le découpage de cette fonction en sous-fonctions a permit une meilleure lisibilité.

draw : Cette fonction vérifie simplement si la surface désignée par *draw_house2* est libre. Si c'est le cas, elle appelle placeHouse

placeHouse : Cette fonction reserve un espace sur la carte matricielle pour l'emplacement d'une maison. Pour ce faire, par convention, une valeur particulière est affectée à la case supérieure gauche de l'emplacement, chaque autre case de la maison reçoit une valeur particuière "occupée".

place_house : Cette fonction va de pair avec la fonction *createSpriteRGB* qui agit en deux temps : création de l'image au format *BMP* de la ville sans maison, puis l'ajout des maison. C'est à la seconde étape qu'est appelée *place_house*. Celle-ci récupère chaque pixel d'une image *BMP* chargée en fonction du type de maison et les applique sur la carte de la ville.

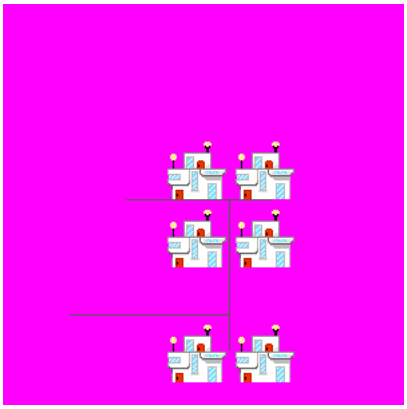


FIGURE 12 – Exemple de résultat à l'issue de l'exécution de *createSpriteRGB*

Les dernières fonctions ajoutées sont les suivantes :

placeSprite : Colle un sprite au format *BMP*, sur une autre image au format *BMP*, en considérant les pixels transparants. Étant donné que le format *BMP* ne supporte pas la transparence, (à l'inverse du format *PNG*), la convention que nous avons choisi est la suivante : un pixel est considéré transparent si son code RGB est 255,0,255. Tous les pixels de couleur *(255,0,255)* seront donc ignorés lors de la copie de l'image source vers la destination.

drawName : Transforme une chaîne de caractère de longueur *len*, en image au format *BMP*. Chaque caractère de l'alphabet est dessiné en noir sur fond rose (255,0,255) dans un fichié *BMP* de dimension 7x7. *drawName* initialise donc une *SDL_Surface* de dimention 7**len*x7, dans laquelle chaque pixel non roses (255,0,255) de chaque caractère est copié. Cette *SDL_Surface* est ensuite sauvegardée au format *BMP*.

FIGURE 13 – Exemple de résultat à l'issue de l'exécution de *drawName*

Il avait été annoncé lors de la dernière soutenance que la génération des noms des villes serait fait à l'aide d'un réseau de neurone. Il s'avère que l'utilisation de tels algorithmes pour le langage est autrement compliqué que de traiter des signaux binaire tels que ceux nous avons pu manipulés lors du projet OCR. Léa et Gauthier ont donc convenu d'une méthode alternative pouvant garantir des résultats satisfaisants. Pour cela ils ont établit certaines règles bannissant ou écourageant certains motif. Celles-ci fonctionnent selon des probobilités d'apparition des lettres. Pour les explications détaillées, confer la partie de Léa.

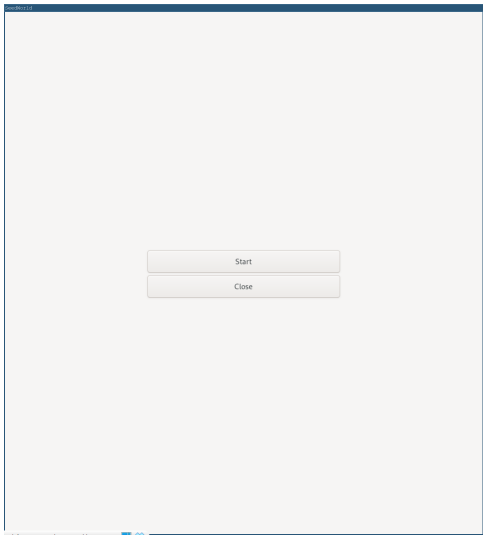
La mise en commun du projet en a été la partie la plus cruciale. Il s'est agi dans un premier temps de fusionner les codes de Léa et de Gauthier qui fonctionnent en symbiose. Pour ce faire une fichier commun *city_build.c* a été créé, utilisant leurs fonctions ensemble. Dans un second temps les codes de Léo et d'Arnaut ainsi que ce de Léa et Gauthier on été assemblés. La "super" fonction contenue dans *city_build.c* a été injécté à sont tour dans le fichier Perlin.c afin d'ajouter les fonctionnalités développées par Gauthier et Léa dans la procédure finale.

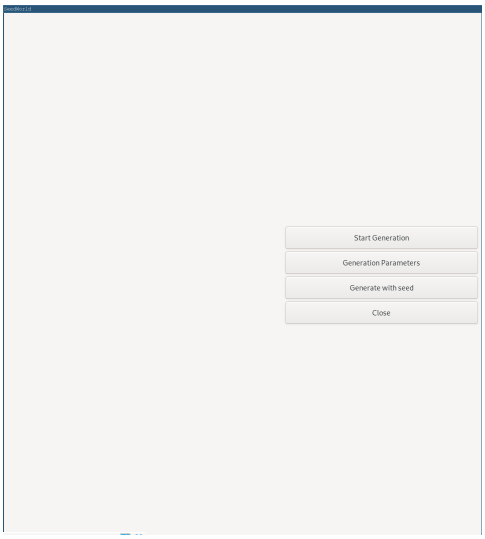
6 Interface

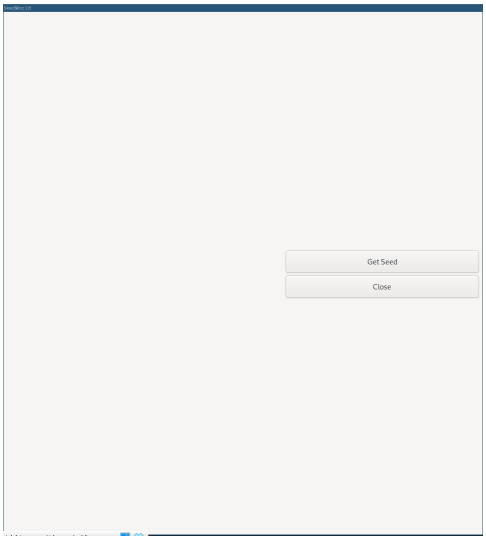
6.1 Première soutenance

6.1.1 Léo Tripier

C'est essentiellement sur cette partie que Léo a travaillé pour cette soutenance. Il a codé le squelette de notre interface. La structure principale repose sur un fichier xml en *.glade* qui sert de base pour créer l'interface.
Pour écrire ce fichier il s'est inspiré de celui qu'il avait déjà fait pour le projet OCR du S3. Il a cependant réécrit pratiquement entièrement le fichier car il souhaitait que la mise en forme de cet interface soit différente. Pour l'interface de ce projet, il utilise des objets appellés "GtkGrid" qui lui permettent de "stocker" des boutons et des les disposer comme il le souhaite dans la fenêtre de l'interface. Pour l'instant il a créé 3 GtkGrid qui seront les 3 principales. Chacunes contenant des boutons différents. Le gros avantage d'utiliser les GtkGrid c'est que cela minimise grandement les lignes de codes dans le programme principal. En effet grâce à ces outils il suffit (dans le programme principal) de "connecter" l'un de ces objets à la fenêtre pour qu'il s'affiche, ou de le "déconnecter" pour qu'il ne s'affiche plus. Cela ne prend donc que 2 ligne de code pour modifier l'affichage de la fenêtre (une pour déconnecter l'ancienne Grid, une autre pour connecter la nouvelle).
Cependant cela nécéssite de bien écrire le fichier en *.glade* qui sert de base afin que tout soit propre. Léo a préféré faire ce choix car cela rend le code beaucoup plus lisible étant donné qu'il ne faut que quelques lignes pour changer la GtkGrid qui sera affichée. Ce qui lui pris le plus de temps est donc bien entendu l'écriture du fichier *"interface.glade"* mais surtout la documentation sur la manière dont s'écrit un fichier de ce type. Il est en effet nécéssaire de connaitre les différents objets mis à disposition par GTK ainsi que les différentes propriétés qui leurs sont propres.
Il lui a également fallut tester beaucoup de fois comment cela rendait car certaines propriétés sont prioritaires par rapport a d'autres, bien que cela ne soit pas toujours explicite dans la documentation GTK. De plus il faut faire attention à ce qu'une nouvelle propriété ne vienne pas annuler une ancienne. Une fois l'équilibre trouvé et la mise en forme satisfaisante il ne reste plus qu'a ajouter les quelques lignes de code évoquées précédemment. Voici quelques images du squelette de l'interface :







Pour l'explication des différents boutons que vous avez pu observer sur ces images :
— Close : ferme la fenêtre
— Start Generation : lance la génération et affiche la grid suivante
— Generation Parameters : affiche les réglages de la génération (à implémenter)
— Generate with seed : permet de donner une seed au générateur afin qu'il génère une map précise (à implémenter).
— Get Seed : affiche la seed de la map qui a été générée (à implémenter)

Il reste donc encore beaucoup de chose à faire. L'objectif pour la prochaine soutenance sera d'implémenter la page des réglages de la génération ainsi que d'embellir un peu cette interface qui est encore très terne.

6.1.2 Arnaut Leyre

Pour cette soutenance l'interface n'a pas été une priorité car elle est vouée à s'adapter au reste du projet. Ainsi seules les fondations ont été posées. Pour les soutenances futures il est prévu de consacrer davantage d'effort et de fonctionnalité. Pour cela il sera permis à l'utilisateur d'influencer sur la génération du monde. Notamment au travers du choix de la seed de génération et de diverses tailles qui seront disponibles.

6.2 Deuxième soutenance

6.2.1 Léo Tripier

Depuis la dernière soutenance Léo a ajouté de nouveau bouton à l'interface. Deux boutons de type *GtkScaleButton*, qui permettent à l'utilisateur de choisir le pourcentage d'océan sur la carte ainsi que la taille des biomes. Le troisième bouton permettait à l'origine de revenir en arrière dans l'interface. Cependant Arnaut ayant repris l'interface depuis, le bouton ne sera sûrement pas garder dans la nouvelle version de l'interface. Les deux autres boutons

eux seront utilisés dans la nouvelle version.
Léo a également commencé à connecter l'interface avec le reste du code. En effet la création d'une nouvelle structure *UserData* contenant les différentes variables relatives au paramètre choisi par l'utilisateur.

```
typedef struct UserData
{
  gdouble b_size;
  gdouble r_ocean;
  int mwidth;
  int mheight;
  int w_type;

} UserData;
```

FIGURE 14 – Structure UserData

Cette structure permettra donc de conserver toutes les variables qui seront ensuite utilisées lors de l'appel des fonctions principales de notre code. Parmi ces variables on trouve *(gdouble)b_size* qui permet de stocker la valeur choisie pour la taille des biomes, *(gdouble)r_ocean* qui permet de stocker le pourcentage d'océan souhaité sur la carte. Actuellement ce sont les deux variables qui sont adaptées en fonction des choix de l'utilisateur car les deux boutons ajoutés sont directement reliés à ces variables. Les trois autres : *(int)mwidth* , *(int)mheight* , *(int)w_type* n'ont pas encore leurs boutons associés donc ne sont pas réellement utilisés pour l'instant.
Lors du lancement de l'interface des valeurs sont choisies aléatoirement pour ces variables au cas où l'utilisateur souhaiterait générer une carte totalement aléatoire sans choisir de paramètres particuliers. Pour l'instant, les variables relatives à la taille de la carte (*mwidth* et *mheight*) sont fixées par défaut à 256 pour faciliter les tests.
Léo a également commencé avec l'aide d'Arnaut à tenter d'utiliser le paramètre de proportion d'océan sur la carte. Cependant les tests ne furent pas concluant, il faudra retravailler sur cela pour la soutenance finale. Le problème semble venir de la formule utilisée, Arnaut et Léo se documententeront pour trouver quelle formule utiliser afin de manipuler cette proportion d'océans.
D'après les informations issues des premières recherches il semblerait qu'un calcul de l'écart-type des valeurs de la matrice représentant la carte afin de la combiner avec le pourcentage souhaité soit une bonne base. Il faudra tout de même approfondir les recherches et les tests effectués.

6.2.2 Arnaut Leyre

Pour ce qui est de l'interface il y a eu beaucoup de changement dans la mise en page mais le squelette du code reste inchangé. Toutes les différentes options de l'application ont été rassemblées sur une seule fenêtre qui est divisée en trois parties :

— Le header sur la partie haute de l'écran qui s'adapte à la taille de la fenêtre avec le titre de l'application et les boutons qui gèrent la fermeture de l'application, la réduction de la fenêtre et la mise en plein écran seulement si le système d'exploitation le support.

— La liste des options à gauche qui permet d'accéder à toutes les fonctionnalités déjà implémentées. Pour cela divers objets sont utilisés tel que des zones de texte, des boutons et des scales.

— Enfin tout le reste permet d'afficher l'image résultat.

Par-dessus cela nous avons réalisé beaucoup de test différent pour ajouter des contrôles plus intuitifs. Nous avons donc travaillé sur l'implémentation d'une barre de texte permettant de récupérer une chaine de caractère qui permet de récupérer une seed ainsi que d'afficher celle générait. Parallèlement nous avons commencé à travailler sur les radios Buttons. C'est boutons connecter permette la sélection d'une option parmi une palette de chois.

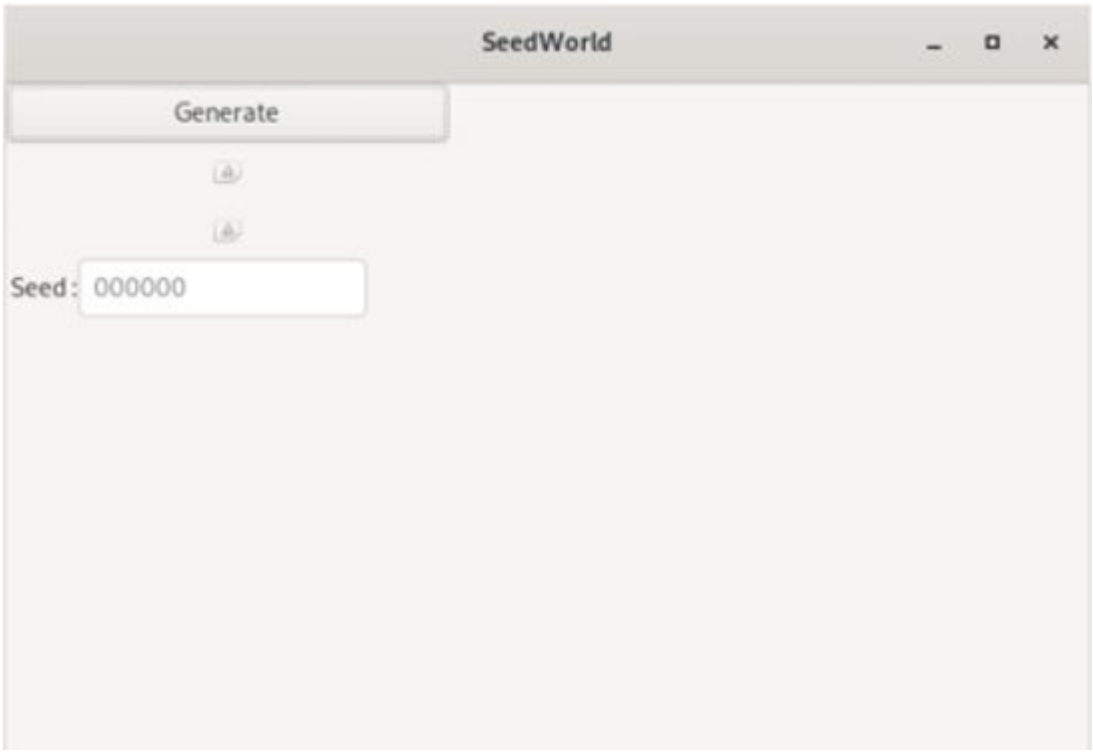


FIGURE 15 – inter

6.3 Soutenance finale

6.3.1 Arnaut Leyre

7 Site web

7.1 Léa Cruciani

7.1.1 Première soutenance

Le site web a été réalisé avec html et mis en page avec css. Léa a décidé de réaliser une barre de navigation fixe positionnée à gauche de la page web pour permettre à l'internautre de passer facilement d'un onglet à un autre sans avoir à retourner au début du site. La barre de navigation est composée de plusieurs onglets :

— Accueil
— Soutenances
— Graphismes
— Progression

L'accueil sera la première page rencontrée par l'internaute une fois sur le site. Il faut qu'elle le renseigne sur l'équipe M.D.W.S, ainsi que bien évidemment sur le logiciel *SeedWorld*. Pour ce faire, plusieurs balises ont été écrites dans la page html *nav* qui sera la barre de navigation à gauche, *article* qui est au centre de la page, il est constitué d'une description du projet *SeedWolrd*, d'une définition de la génération procédurale. À la droite se trouve une brève présentation de l'équipe M.D.W.S, avec le logo(à venir) qui est associé.
L'onglet *Soutenances* permet d'accéder aux téléchargements du rapport pour chaque soutenances, les liens de téléchargements seront actualisés en fonction des soutenances à venir.
L'onglet téléchargement permettra de télécharger *SeedWorld*.
L'onglet progression servira à indiquer l'avancement des tâches de chacun des membres de l'équipe en pourcentage.
L'onglet graphismes montrera un aperçu des bâtiments, des routes, et des biomes dans le logiciel. Il sera actualisé en fonction des dessins réalisés.

7.1.2 Deuxième soutenance

Concernant le site web, l'onglet progression a été actualisé avec l'affichage des pourcentages avec une barre de progression circulaire pour chaque différente tâche que doit accomplir l'équipe M.D.W.S. Cela permettra à l'internaute de mieux visualiser où en est la réalisation du logiciel. L'onglet soutenance a également été modifié pour pouvoir rendre possible le téléchargement du rapport de soutenance 2 en fichier *.pdf*. Enfin le site a été hébergé sur gitHub Pages (`https://myochi.github.io/seedworld/`).

7.1.3 Soutenance finale

Pour cette dernière soutenance l'onglet *Téléchargement* a été actualisé avec l'ajout de deux liens permettant le téléchargement du *.pdf* du mauel d'installation et du Logiciel *SeedWorld.exe*. L'onglet *Progression* a été mis à jour avec l'ajout des tâches réalisées par l'équipe.

8 Graphismes

FIGURE 16 – Auteure Léa

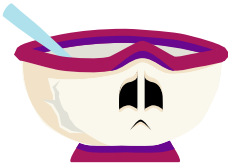


FIGURE 17 – Auteure Léa

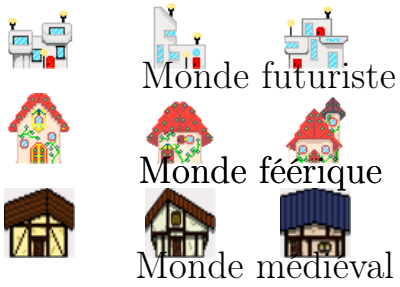


FIGURE 18 – Auteure Léa et Gauthier

9 Les joies et les peines

9.1 Léa

9.1.1 Les joies

Ce projet fût fort intéressant à réaliser. Le groupe était soudé et communiquait énormément ce qui a rendu la mise en commun moins difficile que prévu. Avec la manipulation omniprésente des pointeurs et des matrices, Léa a pu davantage apprendre à manipuler ces structures et sent une nette amélioration comparée au S3. Ce projet aura permis à Léa de découvrir la génération procédurale et d'améliorer ses compétences dans le langage *C*. Elle en retient de très bons souvenirs.

9.1.2 Les peines

La tâche qui est restée malgré tout la plus complexe a tout de même était la mise en commun et notamment la recherhe dans chacun des codes de l'autre personne avec qui les codes ont été liés des bugs et de l'origine de certains *Segmentation fault*. La fonction qui aura été la plus longue à débugger et à coder aura été *build_roads()* car il y avait beaucoup de paramètre à prendre en compte pour pouvoir créer un réseau routier de manière aléatoire mais sensé(éviter les aglutinements, les routes qui se superposent...). Mais une fois arrivé au bout du tunnel, la personne est satisfaite !

10 Bilan

10.1 Arnaut Leyre

Cette soutenance a permis de finir d'implémenter la liste des options à ajouter à l'interface. En termes de fonctionnalité le projet est assez complet. L'interface semble explicite. Bien que les ajustements automatiques puissent être surprenant au début, il est facile de s'y adapter. Avec plus de temps il aurait été intéressant de rajouter des flèches pour explorer les plus grandes tailles depuis l'application.

10.2 Gauthier Marchetti

La conclusion générale de ce projet est très positive. Le groupe et ses dynamiques de travail ont été très intéréssant et motivant. Le projet en lui même s'est révélé être un vrai défi et Gauthier a appris beacoup de choses.

10.3 Léa Cruciani

Ce projet aura permis à Léa d'améliorer ses compétences en C et de se familiariser davantage avec la manipulation des pointeurs qui posait problème dans un premier temps. L'équipe M.D.W.S. était organisée en fonction des différentes parties à réaliser et il y avait une bonne communiquation. Même si la gestion du temps était un peu limite à certains moments, Léa a su s'organiser davantage comparé aux projets réalisés antérieurement. Même si le résultat n'est pas parfait Léa est fière de ce que son équipe et elle ont pu accomplir.

10.4 Léo Tripier

Le bilan est très positif pour Léo. Ce projet fut très intéréssant, tant dans son thème et son développement que dans le groupe de travail. Léo n'avait jamais travaillé avec aucun des membres de ce groupe, ce fut intéréssant de decouvrir de nouvelles façons de travailler avec de nouveaux coéquipiers. De plus tout le travail effectué pour ce projet fut très enrichissant. En partant du bruit de perlin pour générer des structures aléatoires mais cohérentes, qui forme une base très solide pour pouvoir ensuite maitriser par exemple l'ajout des biomes et de la gestion de la proportion de ceux-ci, ce projet m'as permis de beaucoup apprendre.

11 Annexes

Références

[1] Page wikipédia du thème du projet :
`https://fr.wikipedia.org/wiki/Génération_procédurale`, consulté 1 mars 2021

[2] Page Wikipédia du bruit de Perlin :
`https://fr.wikipedia.org/wiki/Bruit_de_Perlin`, consulté 3 mars 2021

[3] Page de renseignement sur le bruit de Perlin :
`https://web.archive.org/web/20080724063449/http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`, consulté 3 mars 2021

[4] Document étudiant sur le sujet :
`https://constellation.uqac.ca/4559/1/Prin_uqac_0862N_10451.pdf`, consulté 3 mars 2021

[5] Archives ouvertes sur le sujet :
`https://tel.archives-ouvertes.fr/tel-00841373/document`, consulté consulté 3 mars 2021

[6] Documentation GTK :
`https://developer.gnome.org/gtk3/stable/index.html`