

Rapport de stage

Génération procédurale de terrain

Réalisé par : **Adrien Galland**

Encadrant : **François Brucker**

Lieu du stage : **LIS, Centrale Méditerranée**

Période du stage : **Juillet 2025**

Sommaire

1 Remerciements	2
2 Présentation du laboratoire	2
3 Introduction	2
4 Pourquoi s'intéresser au bruit de Perlin ?	3
5 Comment est créé le bruit de Perlin ?	3
6 Amélioration de notre algorithme	8
7 Les premières images de la carte	9
8 Amélioration de la carte	10
9 Ajout des rivières	12
10 Modification de la hauteur	14
11 Interface graphique	15
12 Conclusion	15
13 Bibliographie	16

1 Remerciements

Je tiens tout d'abord à remercier M. Brucker, mon encadrant de stage, pour m'avoir si bien guidé et accompagné durant ce mois de stage et pour m'avoir proposé ce sujet très intéressant qui m'a captivé du début à la fin.

Je tiens également à remercier M. Préa pour ses discussions et pour m'avoir donné de nouvelles pistes de recherche. Enfin, merci à M. Deschamps pour m'avoir autorisé à réaliser ce stage.

2 Présentation du laboratoire

J'ai réalisé mon stage au LIS (Laboratoire d'Informatique et des Systèmes), sur le campus de Centrale Méditerranée. Le LIS est un laboratoire rattaché au CNRS, à l'Université d'Aix Marseille (AMU) et à l'Université de Toulon (UTLN). Il est également partenaire de l'école Centrale Méditerranée.

Le LIS a pour objectif de mener des recherches autant fondamentales que appliquées dans les domaines de l'informatique, du signal, de l'image et de l'automatique. Son organisation est structurée en 20 équipes de recherche réparties en 4 pôles.

3 Introduction

Dans de nombreux jeux vidéo, le "monde" dans lequel l'histoire se déroule est amené à changer à chaque nouvelle partie, créant de manière aléatoire de nouveaux terrains et cartes du monde. Le but de mon stage était de comprendre comment ces terrains sont créés aléatoirement tout en étant réalistes puis de créer un programme Python de génération procédurale de terrains à l'image de ce que font les jeux vidéo.

Lors de mon premier entretien, M. Brucker m'a donné comme point de départ le bruit de Perlin en me disant que ce procédé était utilisé dans de nombreux jeux vidéo. Il m'a ensuite présenté des idées de ce qu'il voulait que je réalise : je devais être capable de créer un générateur procédural de terrains qui permettrait (en fonction de plusieurs paramètres réglables) d'obtenir différents types de cartes telles que des îles plus ou moins grandes avec du relief plus ou moins élevé. Il m'a aussi parlé de la possibilité de pouvoir modifier la hauteur du terrain (dit "terraforming") puis de rajouter des rivières. Tout cela devait idéalement être présent dans une interface graphique simple à utiliser pour qu'un utilisateur puisse s'en servir sans explication.

J'ai consacré les premiers jours de mon stage à comprendre ce qu'était le bruit de Perlin et en quoi il pouvait m'être utile, puis après avoir confirmé son utilité je me suis occupé de coder sa génération en Python, puis de fabriquer une carte à partir de ce bruit. J'ai ensuite ajouté les fonctionnalités attendues telles que l'ajout de rivières, le terraforming (modélage du terrain) et une interface utilisateur pour permettre à toute personne d'utiliser mon programme sans documentation. Chacune de ces étapes occupera une partie de mon rapport.

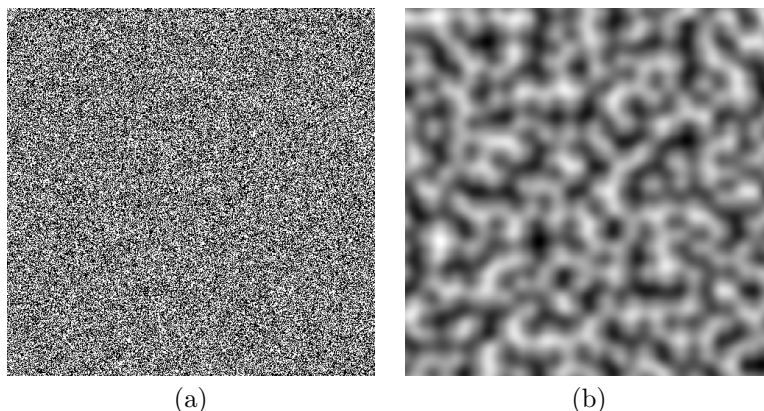


Figure 1: (a) Image de bruit uniforme entre 0 et 1, où chaque pixel a été généré indépendamment de ses pixels voisins (bruit blanc). On remarque qu'aucune structure caractéristique n'est visible. (b) Image de bruit de Perlin, on remarque la présence de motifs, qui peuvent faire penser à des motifs que l'on retrouverait dans la nature.

4 Pourquoi s'intéresser au bruit de Perlin ?

Le but de ce stage étant de créer un algorithme de génération de terrains et que le terrain généré soit différent à chaque nouvelle utilisation. Supposons que nous voulions générer une carte de terrain comportant 1000×1000 pixels. Une première idée pour que cette carte soit différente à chaque nouvelle exécution du programme est de générer une image où l'intensité en chaque pixel est choisie aléatoirement entre 0 et 1 (cf. Figure 1.a). Ces valeurs serviront alors de base pour construire notre carte d'altitude. Cependant, nous voyons ici que chaque pixel a été généré de façon indépendante de ses pixels voisins (on parle de bruit blanc) et nous ne voyons apparaître aucune structure contrairement à ce que nous observons dans un paysage réel.

C'est ce problème que Ken Perlin a cherché à résoudre en 1985 lorsqu'il inventa le bruit de Perlin (cf Figure 1.b).

Tout en étant toujours aléatoire, le bruit de Perlin produit des images sur lesquelles on peut voir des motifs, des régularités, des variations rapides de l'intensité, mais également des zones où l'intensité reste à peu près constante. C'est pourquoi nous utiliserons du bruit de Perlin pour générer notre carte car il est capable de générer des structures naturelles très convaincantes.

5 Comment est créé le bruit de Perlin ?

Pour comprendre comment est créé le bruit de Perlin, intéressons-nous tout d'abord à son cousin proche : le bruit de valeur (ou "value noise" en anglais).

Pour bien comprendre l'objectif, prenons l'exemple d'une montagne. Si nous

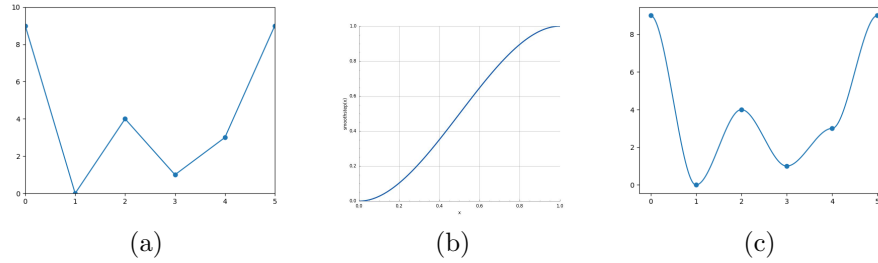


Figure 2: (a) Interpolation linéaire entre 6 valeurs. (b) Fonction d’assouplissement Smoothstep qui varie de 0 à 1 de façon non linéaire. (c) Interpolation avec fonction Smoothstep.

considérons plusieurs points assez proches sur cette montagne, leur altitude sera plutôt proche. Si l’on souhaite générer ces altitudes de manière aléatoire, il est donc nécessaire d’imposer que la plupart du temps l’altitude d’un pixel soit proche de l’altitude de ses pixels voisins. Comme nous l’avons vu sur la figure 1.a, ce n’est pas le cas si l’on génère directement ces altitudes à partir d’un bruit blanc, c’est-à-dire d’un bruit où l’altitude de chaque pixel est générée sans prendre en compte ses pixels voisins.

Commençons tout d’abord par nous intéresser au cas à une dimension. Pour cela, créons un axe numéroté de sorte que l’on puisse attribuer à chaque pixel une abscisse. Sur les exemples suivants, nous avons choisi des abscisses variant de 0 à 5. A chaque abscisse entière, nous lui associons un nombre aléatoire tiré uniformément entre 0 et 1 qui donnera la valeur b du bruit à cette abscisse (cf valeurs marquées par un rond sur la Figure 2.a).

Mais qu’en est-il des pixels intermédiaires tel que par exemple le pixel situé à l’abscisse 1.5 ? Il nous faut un mélange entre les deux valeurs aléatoires voisines, dans notre exemple $b(1)$ et $b(2)$. Nous allons dans un premier temps nous contenter d’une approche linéaire et effectuer une interpolation linéaire entre ces 2 valeurs. Cela revient à définir :

$$b(x) = b^- + dx \times (b^+ - b^-) \quad (1)$$

où $b^+ = b(\lceil x \rceil)$ et $b^- = b(\lfloor x \rfloor)$ et $dx = x - \lfloor x \rfloor$. La quantité dx est donc la progression de x par rapport au nombre entier précédent (dans notre exemple $dx = 0.5$). Nous appellerons cette fonction lerp (pour "linear interpolation") et qui s’écrira $\text{lerp}(b(\lfloor x \rfloor), b(\lceil x \rceil), dx)$. Nous pouvons donc appliquer cela à n’importe quel pixel de coordonnée x : on obtient un graphique où les points situés aux abscisses 0, 1, 2, ... sont reliés par des segments (cf Figure 2.a).

Nous avons bien satisfait nos demandes : les pixels qui sont suffisamment proches les uns des autres ont des valeurs proches. De plus, le fait que les valeurs aux abscisses entières soient générées aléatoirement permet qu’on obtienne un résultat différent à chaque génération. C’est ce qu’on appelle le "bruit de valeur". Cependant, ce n’est pas encore satisfaisant à cause des fortes vari-

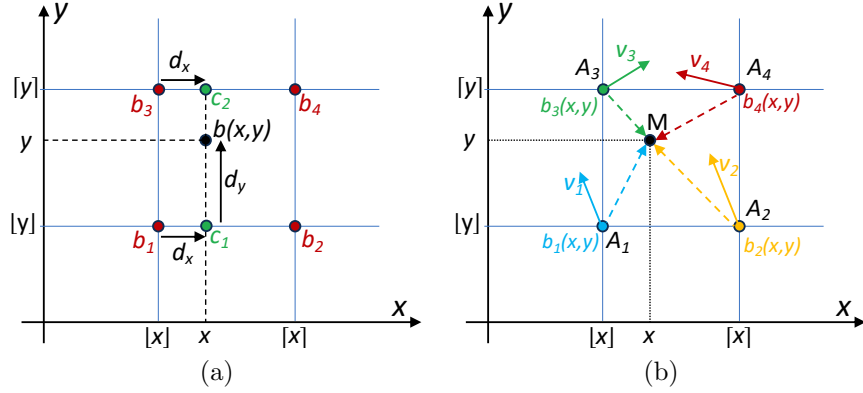


Figure 3: (a) Schéma de l'interpolation bilinéaire. (b) Schéma du fonctionnement de la génération du bruit de Perlin.

ations de pente au niveau des abscisses entières. Heureusement, il est possible d'apporter une amélioration. Pour l'instant nous avons utilisé une interpolation linéaire. Si maintenant nous appliquons une fonction à la quantité dx de l'équation (1), il est possible de rendre cette interpolation non linéaire et d'obtenir des interpolations plus lisses au niveau des abscisses entières. On appelle ces fonctions des fonctions d'assouplissement. La plus populaire s'appelle la fonction "smoothstep" et vaut :

$$\text{Smoothstep}(x) = 3x^2 - 2x^3 \quad (2)$$

Son avantage est que sa progression n'est plus linéaire et que sa dérivée vaut 0 en 0 et en 1 (cf Figure 2.b).

En utilisant $\text{smoothstep}(dx)$ au lieu de simplement dx dans notre interpolation de l'équation (1), notre bruit de valeur sera une fonction qui sera continue et dont la dérivée est également continue (cf Figure 2.c).

Cela termine notre implémentation du bruit de valeur en une dimension. Mais pour générer une carte, du bruit unidimensionnel n'est pas suffisant. Intéressons-nous donc au cas à 2 dimensions pour pouvoir créer des textures. Notre fonction de bruit dépendra donc maintenant d'une valeur x et d'une valeur y , et sera noté $b(x, y)$ où x et y sont les coordonnées dans l'image. Sur les exemples suivants, nous avons choisi de faire varier l'abscisse x de 0 à 5 et l'ordonnée y de 0 à 5 lorsque l'on parcourt l'image.

Il ne nous reste plus qu'à appliquer le même procédé que précédemment, mais en 2 dimensions. Afin de trouver les coordonnées entières les plus proches de notre pixel, nous prendrons les valeurs entières supérieures et inférieures des composantes x et y . Pour cela, nous ferons d'abord une interpolation entre les 2 valeurs des coins inférieurs de la grille sur l'axe x puis entre les 2 valeurs des coins supérieurs de la grille toujours sur l'axe x , puis nous ferons une dernière interpolation entre les 2 valeurs obtenues sur l'axe y .

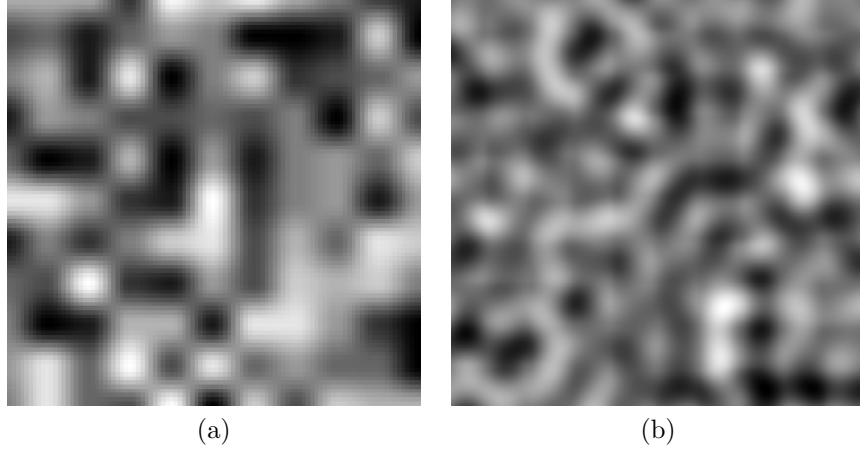


Figure 4: (a) Image de bruit de valeur. (b) Image de bruit de Perlin.

Cela nous donne la formule :

$$b(x, y) = \text{lerp}(c_1, c_2, dy) \quad (3)$$

avec

$$c_1 = \text{lerp}(b_1, b_2, dx) \text{ et } c_2 = \text{lerp}(b_3, b_4, dx) \quad (4)$$

et où $dx = x - \lfloor x \rfloor$ et $dy = y - \lfloor y \rfloor$ et où les valeurs b_1 , b_2 , b_3 et b_4 sont les valeurs aux 4 pixels entiers autour du pixel (x, y) (voir figure 3.a).

Ce processus s'appelle l'interpolation bilinéaire. Comme précédemment, nous avons également choisi d'utiliser dans la fonction lerp la fonction smoothstep afin d'obtenir une fonction plus lisse. Si nous étendons ce procédé à l'ensemble de la grille, nous pouvons maintenant visualiser notre bruit de valeur en 2 dimensions sur la figure 4.a.

Comme on peut le voir, cela nous permet d'obtenir une image aléatoire faisant apparaître des motifs et des zones à peu près constantes, contrairement à l'image du bruit blanc. Cependant, les motifs restent très orientés sur les directions horizontales et verticales.

Maintenant que nous avons pu créer du bruit de valeur, nous pouvons enfin passer à la génération du bruit de Perlin. Si nous avons dit plus tôt que le bruit de Perlin et le bruit de valeur étaient cousins, c'est parce qu'ils sont tous les deux basés sur cette génération aléatoire sur des valeurs (x, y) entières. Ils construisent de plus leur sortie de la même manière, c'est-à-dire avec l'interpolation des valeurs des 4 pixels entiers les plus proches. Ce qui les différencie est la manière que ces deux algorithmes ont de générer ces 4 valeurs. En effet, le bruit de Perlin se base sur des vecteurs pour trouver la valeur de chaque pixel entier au lieu de simplement tirer des nombres aléatoires.

La première étape de génération de notre bruit de Perlin consiste donc à associer un vecteur à chaque coordonnées entières de la grille. Il s'agit d'un vecteur à 2 composantes dont chacune est tirée aléatoirement entre -1 et 1.

Ensuite, pour chaque point M de coordonnées (x,y) dont on cherche à calculer la valeur bruitée $b(x,y)$, nous pouvons déterminer ses 4 pixels les plus proches de coordonnées entières. Si l'on note $x^- = \lceil x \rceil$ et $x^+ = \lfloor x \rfloor$ et $y^- = \lceil y \rceil$ et $y^+ = \lfloor y \rfloor$, ces 4 pixels ont pour coordonnées $A_1 = (x^-, y^-)$, $A_2 = (x^+, y^-)$, $A_3 = (x^-, y^+)$ et $A_4 = (x^+, y^+)$. Nous calculons alors pour chacun de ces pixels A_i , avec $i=1,2,3,4$, une valeur $b_i(x,y)$ définie comme le produit scalaire entre le vecteur aléatoire \vec{v}_i associé au point A_i et le vecteur $A_i\vec{M}$. Nous obtenons ainsi une valeur $b_i(x,y)$ pour chacun des pixels entiers voisins du pixel (x,y) considéré (cf Figure 3.b). Ces valeurs dépendent maintenant du point (x,y) , d'où la notation $b_i(x,y)$ et non plus simplement b_i . A partir de ces 4 valeurs, il est alors possible d'appliquer le même processus que pour le bruit de valeur, c'est-à-dire de faire une interpolation entre ces 4 produits scalaires $b_i(x,y)$ exactement comme dans l'équation (3) et (4), mais en utilisant ces nouvelles valeurs $b_i(x,y)$ au lieu de b_i .

En faisant cela pour chaque point de la grille, nous obtenons l'image de la figure 4.b.

Contrairement au bruit de valeur, le bruit de Perlin fait apparaître des motifs sans orientation particulière, ce qui lui donne une apparence plus réaliste et le rend plus adapté pour l'usage que nous voulons en faire. Ce qu'on a obtenu est le bruit de Perlin original inventé par Ken Perlin en 1983. Mais depuis, il lui a apporté quelques améliorations. Passons donc en revue ces améliorations.

Le premier point vient de la manière dont nous générons les vecteurs des pixels entiers. Si nous générons simplement des nombres aléatoires entre -1 et 1 pour les composantes x et y , cela revient à créer un vecteur dans un carré unitaire, ce qui conduit à ce que les vecteurs orientés à $\pm 45^\circ$ ont une probabilité plus grande d'être tirés que les vecteurs orientés à 0° ou 90° . Pour résoudre ce problème, il faut tirer notre vecteur sur un cercle plutôt que dans un carré. Pour cela, il suffit simplement de tirer non pas les composantes du vecteur, mais l'angle α qu'il forme avec l'horizontale. Il ne reste plus qu'à calculer les composantes à partir de cet angle :

$$x = \cos(\alpha)$$

$$y = \sin(\alpha)$$

Avec cette méthode de génération de vecteurs, tous les angles ont par construction la même probabilité d'être tirés.

Un autre problème vient de notre fonction smoothstep. En effet, on peut remarquer dans notre bruit qu'il y a parfois des incohérences au niveau du positionnement de la grille. Cela vient du fait que si l'on trace la dérivée seconde de notre fonction smoothstep, on remarque qu'elle n'est pas égale pour le point 0 et le point 1. Pour résoudre ce problème, nous devons utiliser une fonction différente. Ken Perlin a choisi une fonction du 5e ordre $f(x) = 6x^5 - 15x^4 + 10x^3$ car elle satisfait les exigences de la continuité de la dérivée seconde. On obtient alors une image du bruit de Perlin telle que celle de la figure 1.b.

Avec ces deux changements, nous obtenons l'algorithme amélioré du bruit de Perlin tel que Ken Perlin l'a présenté en 2004 dans le chapitre 5 de GPU Gems,

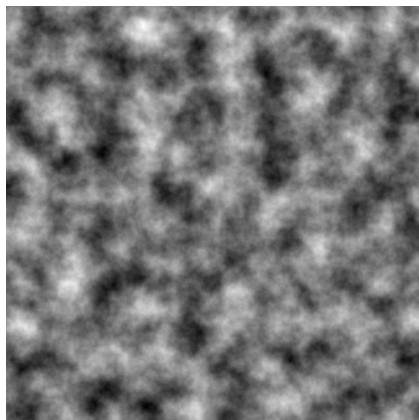


Figure 5: Image de bruit de Perlin de taille 1000×1000 pixels et générée sur 6 octaves, avec une grille initiale où les coordonnées varient de 0 à 10 et une persistance de 0.5 entre chaque octave.

tome 1, qui est un livre technique publié par NVIDIA et présentant des techniques de rendu graphique en temps réel sur processeurs graphiques [3].

6 Amélioration de notre algorithme

Bien que nous ayons du bruit de Perlin, il n'est pas facile de voir la ressemblance avec un terrain réaliste. C'est pourquoi il a été nécessaire de rajouter un paramètre : le nombre d'octaves. Un octave correspond à une génération du bruit de Perlin. Si on a un nombre d'octaves supérieur à 1, il faudra effectuer plusieurs générations du bruit de Perlin puis superposer chaque génération et additionner la valeur prise par chaque pixel dans les différentes couches. Cela consiste à générer une image de Perlin $P_1(x, y)$ où les coordonnées varient de 0 à M , puis une image $P_2(x, y)$ comportant le même nombre de pixels mais où cette fois-ci les coordonnées varient de 0 à $2M$, etc. L'image de Perlin finale est alors obtenue en ajoutant chacune de ces images, pondérées par un coefficient de persistance $\alpha \in [0, 1]$:

$$P(x, y) = P_1(x, y) + \alpha P_2(x, y) + \alpha^2 P_3(x, y) + \dots \quad (5)$$

Notre algorithme de génération de bruit de Perlin a maintenant différents paramètres sur lesquels il est possible de jouer :

- la taille de l'image générée (en nombre de pixels),
- la taille de la grille utilisée pour la génération,
- le nombre d'octaves (un nombre entier valant au minimum 1),
- la persistance (nombre entre 0 et 1).

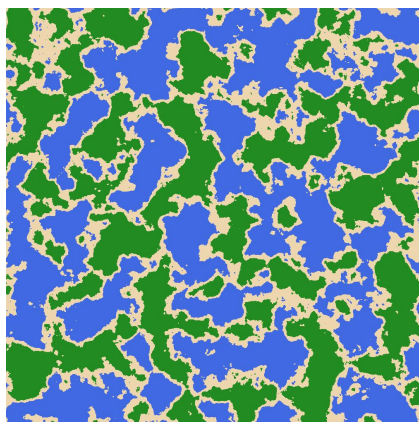


Figure 6: Terrain obtenu à partir d'un bruit de Perlin dans le cas où on a défini 3 classes.

Avec une résolution de 1000, une grille de taille 10 par 10 et 6 octaves de persistance 0.5, nous obtenons une image avec une texture très complexe (cf Figure 5).

7 Les premières images de la carte

Maintenant que nous sommes satisfait par les formes que nous obtenons sur nos images de bruit de Perlin, nous allons nous en servir pour générer nos terrains. Pour cela, nous allons considérer que ces images de bruit de Perlin correspondent à des cartes d'altitudes. Nous pouvons alors directement générer des cartes de terrain en définissant des seuils afin d'attribuer un type de terrain en fonction de l'altitude. En définissant les seuils de la manière suivante :

- si la valeur du pixel est inférieure à 0.5, c'est de l'eau,
- sinon si elle est inférieure à 0.6, c'est du sable,
- sinon c'est de l'herbe.

Nous obtenons l'image de terrain de la figure 6. Bien qu'elle reste extrêmement basique, c'est déjà une très bonne base qui nous permet de confirmer que notre méthode de génération de bruit de Perlin nous donne bien des formes plausibles pouvant servir de contour à une carte. Ce résultat n'étant pas exactement ce que nous voulons, il m'a suffi de rajouter plus de sols différents et d'ajuster leur hauteur afin d'obtenir quelque chose de plus joli. J'ai choisi ces niveaux :

- si la valeur du pixel est inférieure à 0.15, c'est de l'eau,
- sinon si elle est inférieure à 0.18, c'est du sable,
- sinon si elle est inférieure à 0.20, c'est de la plage,

- sinon si elle est inférieure à 0.30, c'est de l'herbe,
- sinon si elle est inférieure à 0.60, c'est de la forêt,
- sinon si elle est inférieure à 0.8, c'est de la montagne,
- sinon c'est de la neige.

Comme nous le verrons plus loin (voir figure 8), cela nous donne un résultat plus satisfaisant avec plus de types de terrains.

8 Amélioration de la carte

Avec la figure 6, nous remarquons que bien que ces premiers résultats soient très prometteurs, la carte en elle-même n'est pas très réaliste, principalement au niveau de la répartition de la terre et de la mer. C'est pourquoi j'ai voulu faire décroître la valeur de chaque pixel en fonction de sa distance au centre de la carte afin d'obtenir une île plutôt qu'un terrain faisant penser à une zone marécageuse [4].

Pour ce faire, j'ai d'abord créé une image de pondération qui vaut 1 au centre de l'image et 0 au bord. Plus précisément, nous créons une image qui a le même nombre de pixels que notre image de bruit de Perlin et qui est égale à l'opposée de la distance au centre de l'image : $d(x, y) = -\sqrt{(x - x_0)^2 + (y - y_0)^2}$ où (x_0, y_0) sont les coordonnées du centre de l'image. La fonction de pondération vaut alors $p(x, y) = (d(x, y) - m) / (M - m)$ où $m = \min(d(x, y))$ et $M = \max(d(x, y))$. On obtient ainsi bien une fonction de pondération qui vaut 1 au centre de l'image et 0 au bord.

Si on affiche cette image de pondération en niveaux de gris, nous obtenons l'image d'un dégradé très progressif (cf Figure 7.a).

J'ai par la suite multiplié chaque pixel du bruit de Perlin par cette fonction de pondération $p(x, y)$ avant de renormaliser les valeurs entre 0 et 1. Cela nous donne le résultat de la figure 7.b.

En réutilisant notre algorithme précédent qui nous a permis de mettre des couleurs à notre carte, nous obtenons une carte de terrain qui ressemble maintenant bel et bien à une île (cf Figure 8.a).

Nous avons maintenant un rendu beaucoup plus satisfaisant. Néanmoins, il n'y a pas forcément de montagne au centre de notre île et si il y en a une, elle est trop petite pour rendre crédible l'apparition d'une telle île. C'est pourquoi j'ai décidé de rajouter un paramètre à l'algorithme qui permet de multiplier les valeurs de la fonction de pondération $p(x, y)$ par une constante h_0 . Ainsi, si ce paramètre vaut 2, les valeurs de $p(x, y)$ seront de 2 au centre et de 0 au bord. Avec une valeur de 2, nous obtenons un type de carte avec des montagnes beaucoup plus élevées au centre (cf Figure 8.b).

Avec ces modifications, nous arrivons donc au terme de notre algorithme de génération de terrain.

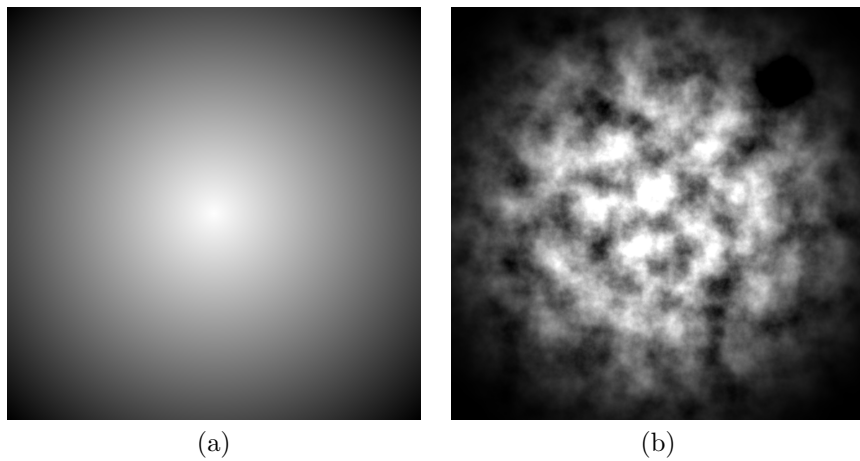


Figure 7: (a) Image de pondération (b) Bruit de Perlin multiplié à l'image de pondération

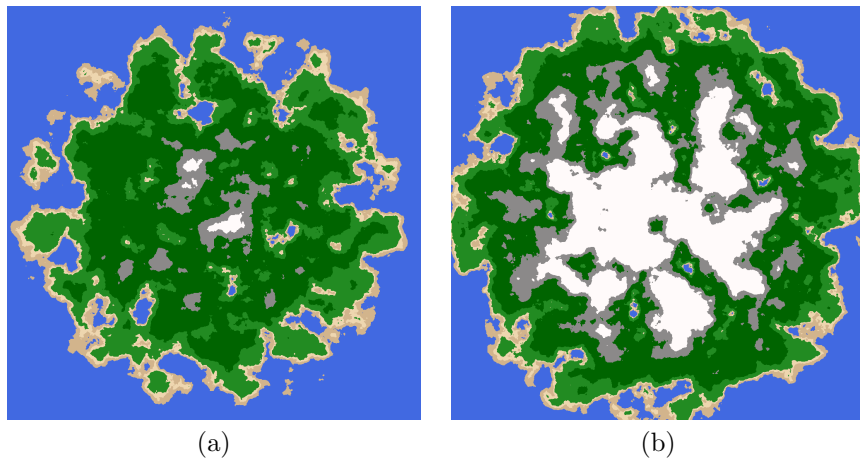


Figure 8: (a) Bruit de Perlin pondéré et coloré. (b) Bruit de Perlin pondéré et coloré avec 2 comme valeur centrale

9 Ajout des rivières

En ce qui concerne l'ajout de rivières, j'ai décidé de laisser l'utilisateur placer les sources. L'objectif est donc de faire "couler" la rivière de la source définie jusqu'à la mer. J'ai pour cela créé un algorithme récursif qui fait aller la rivière (représentée par une liste contenant toutes les cases par lesquelles passe la rivière) vers son pixel voisin d'altitude la plus basse jusqu'à qu'elle rencontre la mer. Le problème avec cette méthode est que parfois la rivière crée un lac et donc la "tête" de la rivière n'a nulle part où aller car de l'eau est présente tout autour d'elle. La première version de cet algorithme s'arrête donc dans ce cas-là :

```
def rivieres(x, y, grille, liste_riviere=None) -> list:
    if grille[y][x] < 0.15:
        return liste_riviere
    if liste_riviere is None:
        liste_riviere = [(x, y)]

    cases_inspectees_x = [x]
    cases_inspectees_y = [y]
    if x != 0:
        cases_inspectees_x.append(x - 1)
    if x < len(grille[0]) - 1:
        cases_inspectees_x.append(x + 1)
    if y != 0:
        cases_inspectees_y.append(y - 1)
    if y < len(grille) - 1:
        cases_inspectees_y.append(y + 1)

    liste_cases = []
    for i in cases_inspectees_x:
        for j in cases_inspectees_y:
            if (i, j) != (x, y):
                liste_cases.append((i, j))

    liste_cases.sort(key=lambda case: grille[case[1]][case[0]])
# Trie les cases par altitude croissante
    print(liste_cases)

    for case in liste_cases:
        if case not in liste_riviere:
            liste_riviere.append(case)
            return rivieres(case[0], case[1], grille, liste_riviere)
    return liste_riviere
```

Cela nous donne des rivières de ce type assez courtes mais avec un tracé assez réaliste (cf Figure 9.a).

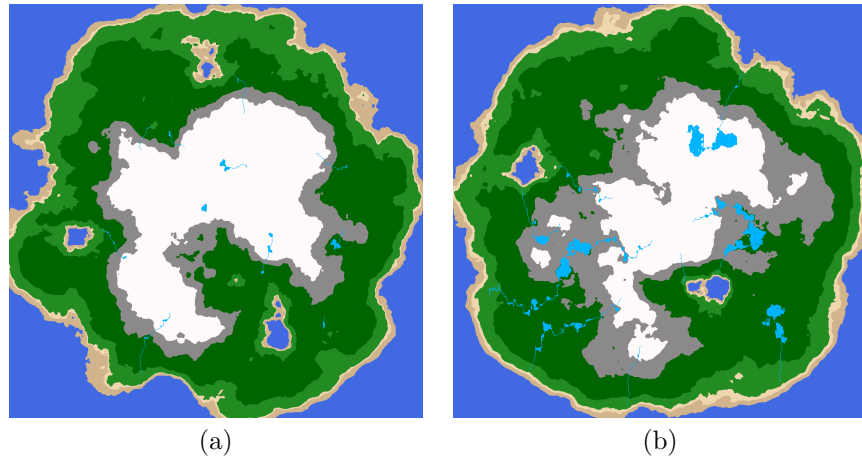


Figure 9: (a) Premier système de génération de rivières s'arrêtant à la rencontre d'un minimum local. (b) Deuxième système de génération de rivières plus complexe qui permet une arrivée certaine des cours d'eau jusqu'à la mer.

Pour pouvoir faire en sorte que la rivière atteigne la mer, j'ai donc changé de manière de procéder. Lorsque la "tête" de la rivière reste bloquée autour d'un lac, alors il suffit de revenir une case en arrière jusqu'à être en capacité de changer de direction. Voici l'algorithme que j'ai créé :

```
def rivières(x, y, grille, liste_riviere=[]):
    if grille[y][x] < 0.15:
        return liste_riviere
    if liste_riviere is []:
        liste_riviere = [(x, y)]
    cases_inspectees_x = [x]
    cases_inspectees_y = [y]
    if x != 0:
        cases_inspectees_x.append(x - 1)
    if x < len(grille[0]) - 1:
        cases_inspectees_x.append(x + 1)
    if y != 0:
        cases_inspectees_y.append(y - 1)
    if y < len(grille) - 1:
        cases_inspectees_y.append(y + 1)

    liste_cases = []
    for i in cases_inspectees_x:
        for j in cases_inspectees_y:
            if (i, j) != (x, y):
                liste_cases.append((i, j))
    # shuffle(liste_cases)
```

```

liste_cases.sort(key=lambda case: grille[case[1]][case[0]])
print(liste_cases)

for case in liste_cases:
    if case not in liste_riviere:
        liste_riviere.append(case)
        print(case[0], case[1])
        return rivieres(case[0], case[1], grille, liste_riviere)
case_invalide = liste_riviere.pop(-1)
liste_riviere = [case_invalide] + liste_riviere
print(liste_riviere)
print(liste_riviere[-1][0], liste_riviere[-1][1])
return rivieres(
    liste_riviere[-1][0], liste_riviere[-1][1], grille, liste_riviere
)

```

Cela nous donne des rivières beaucoup plus réalistes (cf Figure 9.b), bien que les lacs soient peut être légèrement trop présents. Je n’ai malheureusement pas pu accorder plus de temps à trouver un autre algorithme afin d’avoir un meilleur tracé, cette méthode m’aillant déjà pris plus de temps que prévu.

De plus, après m’être documenté, afin d’avoir des tracés beaucoup plus réalistes, il est courant de tracer d’abord des rivières puis d’élever le terrain en conséquence [2]. Mais il aurait fallu pour cela revoir la base même du projet.

10 Modification de la hauteur

Un autre point qu’il a été nécessaire de développer a été la capacité de laisser l’utilisateur modifier le relief de la carte générée. Pour cela, j’ai créé une fonction qui prend comme arguments les coordonnées d’un pixel (celui choisi par l’utilisateur), le rayon d’action, la profondeur/hauteur et une variable booléenne qui permet de savoir si on veut creuser/élever le terrain ou si l’on veut l’aplanir. Ensuite, le programme est assez simple :

- Si on veut niveler le terrain sélectionné, on se contente de parcourir chaque pixel autour du pixel choisi dans le rayon donné et on l’élève ou le rabaisse en fonction de ce qui a été choisi. Afin que le rendu soit réaliste, il faut faire en sorte que le centre du cercle d’application ait une plus grande modification de la hauteur que le bord. On utilise pour cela la fonction smoothstep appliqué à $1 - \frac{\text{distance au centre}}{\text{rayon du cercle}}$ ce qui nous donne un nombre entre 0 et 1 qui sert de coefficient multiplicatif à la modification de la hauteur.
- Si on veut aplanir le terrain, on calcule la moyenne de tous les pixels se trouvant dans le cercle de rayon donné et pour chacun de ces pixels, on augmente sa hauteur de la hauteur entrée en paramètre si elle est inférieure à la moyenne et on la diminue si elle est supérieure.



Figure 10: Interface graphique de mon programme. (a) Ecran d'accueil. (b) Exemple de carte générée et interface utilisateur permettant d'activer l'ajout de rivières ou le terraforming

Il est important de dire qu'à la fin de la modification du terrain, les rivières sont régénérées afin qu'elles s'adaptent au nouveau terrain.

11 Interface graphique

À la fin de mon stage, j'ai développé une interface graphique permettant de pouvoir tester de manière simple et intuitive les différents outils de génération de terrains que j'avais développés. J'ai utilisé pour cela le module pyget [1]. Lorsqu'on lance le programme, une première page s'affiche d'abord nous montrant les valeurs des paramètres actuels de la génération. En dessous, il y a un bouton qui permet de générer la carte. Durant les temps d'attente pendant que le programme effectue des calculs, j'ai fait en sorte qu'un message "LOADING..." s'affiche. J'ai pour cela utilisé la méthode `schedule.once` de `pyglet` qui permet d'attendre quelques instants avant de lancer la fonction appelée, ce qui permet au programme de rafraîchir le contenu affiché à l'écran et donc d'afficher ce message de chargement.

Comme nous pouvons le voir sur la figure 10.b, en haut à droite de l'interface utilisateur se trouvent 2 boutons, l'un pour activer le terraforming (pour modifier l'altitude du terrain) et l'autre pour activer la génération des rivières. Un cadre situé en dessous explique le fonctionnement de chacun de ces modes.

12 Conclusion

Ce stage m'a permis de découvrir un domaine que je ne connaissais pas du tout au départ, celui de la génération procédurale de terrains. J'ai appris à comprendre et à utiliser le bruit de Perlin, puis à m'en servir pour créer un programme capable de générer des cartes réalistes. Tout au long du projet, j'ai ajouté différentes fonctionnalités comme le terraforming, l'ajout de rivières ou encore

l'interface graphique, ce qui m'a permis de découvrir plein d'aspects différents : mathématiques, programmation, visualisation, interaction utilisateur...

J'ai aussi appris à faire face à des problèmes imprévus, comme par exemple le tracé des rivières ou le réalisme du relief, et à adapter mon code pour les corriger. Même si certaines parties auraient pu être encore améliorées avec plus de temps, je suis satisfait du résultat final. Ce stage a été une expérience très enrichissante, et m'a vraiment donné envie de continuer à explorer ce genre de projets à l'avenir.

Certains points pourraient malgré tout être améliorés tels que la génération de rivières qui peut avoir tendance à produire trop de lacs et qui ne suit pas toujours des tracés réalistes. Une méthode envisageable serait de générer de l'érosion grâce aux fonctions de terraforming sur le passage des rivières. Cela donnerait des paysages encore plus réalistes. Il serait également envisageable de créer plusieurs îles, par exemple en générant plusieurs maxima à la fonction de pondération au lieu d'un seul et également d'ajouter différents biomes sur le terrain généré. Un point important serait aussi d'optimiser le programme qui pour l'instant reste assez lent. L'interface graphique pourrait aussi être améliorée en affichant par exemple progressivement une rivière au fur et à mesure de sa génération.

Vous pouvez retrouver les fichiers de mon stage sur mon GitHub : https://github.com/AdrienGalland/Stage_MPCIL1_2025.Generation.procedural_terrain

13 Bibliographie

References

- [1] Pyglet Developers. Pyglet documentation, 2025.
- [2] Jean-David Génevaux, Éric Galin, Éric Guérin, Adrien Peytavie, and Bedrich Beneš. Génération procédurale de rivières et de terrains. In *25èmes Journées de l'Association Française d'Informatique Graphique (AFIG)*, pages 1–10, 2012.
- [3] Ken Perlin. *Implementing Improved Perlin Noise*. NVIDIA, 2004.
- [4] Yvan Scher. Playing with perlin noise: Generating realistic archipelagos, 2016.