Thème: **Jeux et Sports**
22 juin 2023
Encadré par: Thami Elhalkuj, Mustapha Saadaou

Génération procédurale des terrains virtuels dans les jeux vidéos

1

- **Bruit de Perlin**
- **Déplacement du point median** (Midpoint displacement MDP)

Anwar KARDID
MP*
N° CNC: BE016M

Plan

1. Motivation, Problématique
2. Approche 1: Déplacement du point median (MDP)
3. Approche 2: Bruit de Perlin (Perlin Noise)
4. Simulation, confrontation et conclusion

1. Motivation & problématique

Simulations sportives, Jeux videos…

Pour une meilleur experience:

- Diversité du contenu
- Evénements imprévisibles
- ➢ Curiosité et enthousiasme des joueurs

Simulations sportives

Simulations physiques militaires

Jeux videos

Meilleur exemple:

Est-ce que la map de Minecraft est infini ?
Sur Minecraft, la génération procédurale
offre des cartes quasiment infinies.

https://www.minecraft.net/en-us

Aussi:

Il existe 18 446 744 073 709 551 616 **planètes** différentes dans l'univers de **No Man's Sky** (soit **18 trillons** ou 2^{64}).

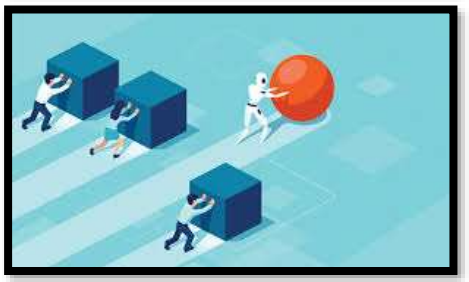
https://www.nomanssky.com/

Problématique ?

- Nécessité des artistes spécialisés,
- Du temps,
- De l'argent…

- Comment aborder les défis de la conception numérique des terrains virtuels dans les jeux

- Et à quel point on peut combiner entre performance, qualité visuelle et optimisation des ressources informatiques?



Avantages

- **Variété et Évolutivité**
  - Expériences uniques et différentes à chaque session de jeu.
  - Maintenir l'intérêt des joueurs le long terme.

- **Optimisation des ressources** :
  - Réduire la taille des fichiers et la consommation de ressources.
  - Pas nécessaire de stocker des éléments préfabriqués, la génération est spontanée.

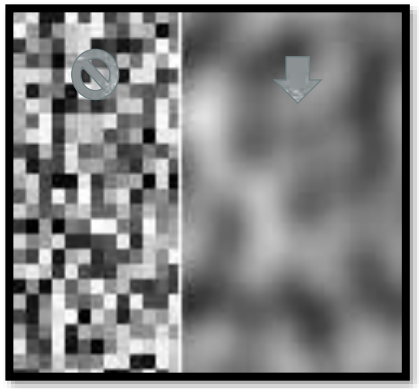
Jeux: Minecraft

Jeux: Sonic Frontiers

Approches:

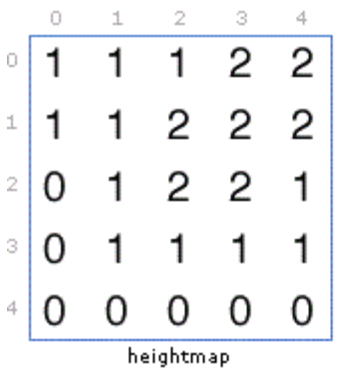
1. Bruit de Perlin:
   - Algorithme développé par Ken Perlin (1982)
   - Professeur d'informatique a New York University
   - Génère des valeurs pseudo aléatoires, avec aspect organique.

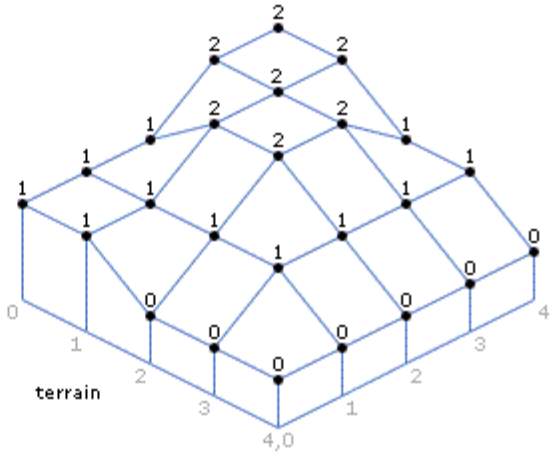
2. Déplacement du point median :
   - Alain Fournier (années 1980)
   - Génération récursive des terrains

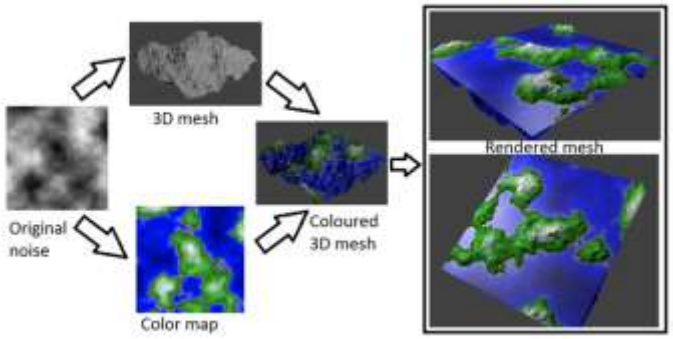


Perlin Noise (rendu local sur machine)

https://en.wikipedia.org/wiki/Ken_Perli







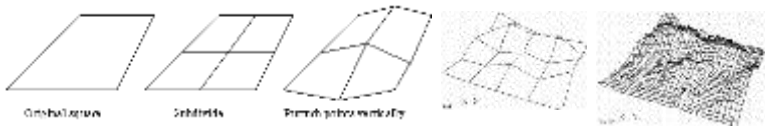
Source: Polynomial methods for fast Procedural Terrain Generation

Préliminaire:

Avant de générer les terrains, nous devons avoir une structure de données pour les stocker.

On s'inspirants des cartes topographiques:

Les matrices de hauteur sont idéales car elles sont légères, rapides à charger, modifiables aisément.



11

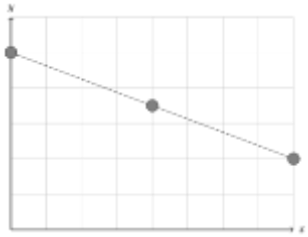
Déplacement du point médian

1) **Procédures**
2) **Résultats**
3) **Complexité**

Initialement on débute avec 2 $(= 2^0 + 1)$ points ayant des hauteurs aléatoires

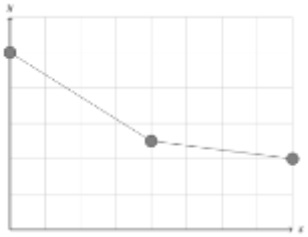
1

Le Point Médian est le milieu de ces deux points, Sa hauteur est la moyenne de leurs hauteurs

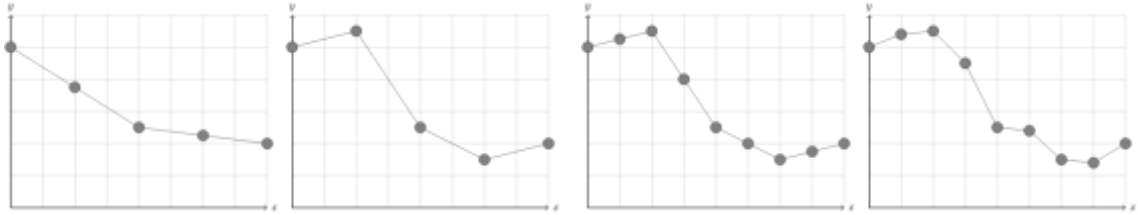


2

Décaler sa hauteur par une valeur aléatoire borné



3



On refait la même procédure récursivement avec les nouveau points et des décalages plus atténués

4

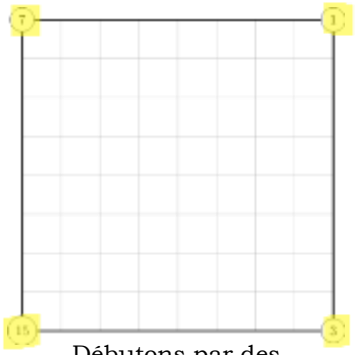
Intuition en 1D

Remarquer que la taille doit être de la forme $(= 2^n + 1)$ afin que les milieux entiers soient définies.

(démonstration en PJ)

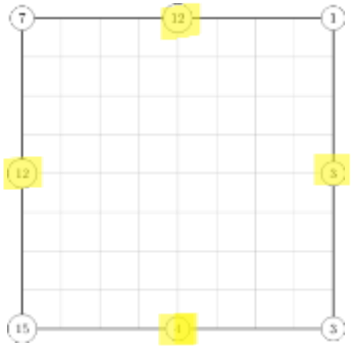
$\mathbf{n} = 2^3 + 1 = 9$

1



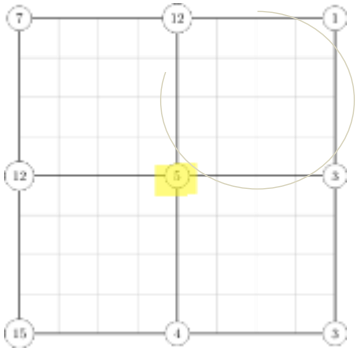
Débutons par des point a hauteurs aléatoires

2



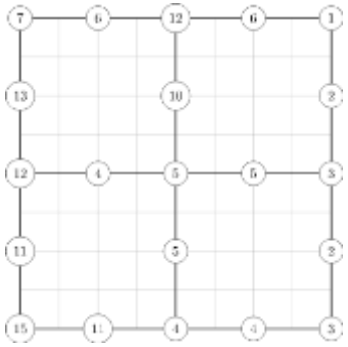
Les médians de chaque coté ont la demi hauteur des deux sommets voisins respectivement, avec un décalage aléatoire.

3



Le centre du carrée a la hauteur Moyenne des quartes sommets, avec un décalage aléatoire.

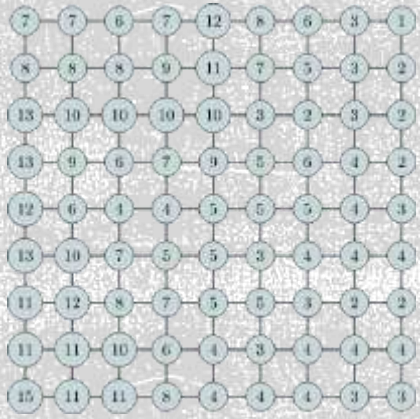
4



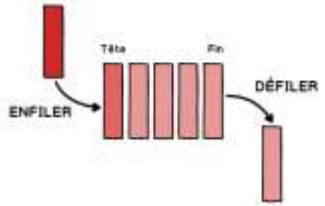
On refait la procédure récursivement avec les nouveaux carrées, avec des décalages moins atténuées

La procédure globalement:

Ainsi on construit notre carte de hauteur

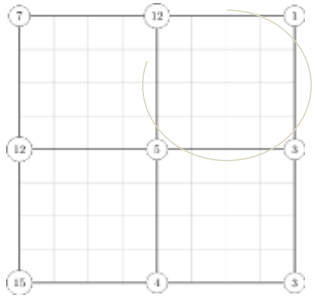


- Un carrée est défini par: deux sommets, et une limite de décalage aléatoire : r
  Ex: (0, 0, n-1, n-1, r) ~(x1, y1, x2, y2, r)
- On utilise une **file** pour stocker chaque carrée découvert, afin de le traiter après.



Algorithm 1 Algorithme de déplacement du point médian

```
1: Début file vide
2: Ajouter le carré initial
3: while la file non vide do
4:     On calcule les hauteurs de chaque médiane des côtés,
5:     puis le centre, avec le décalage du carré actuel
6:     if les médians sont des entiers then
7:         Les 4 carrés prochains sont définis
8:         On les ajoute à la file
9:         Avec une limite décalage / 2
10:    end if
11: end while
12: Fin tant que
```



Déplacement du point médian: Procédure

Preuve de correction de l'algorithme

- **Variant de boucle:**

  La file décroit en taille après un certain rang* **(quant tout les carrées possibles sont ajoutés)**

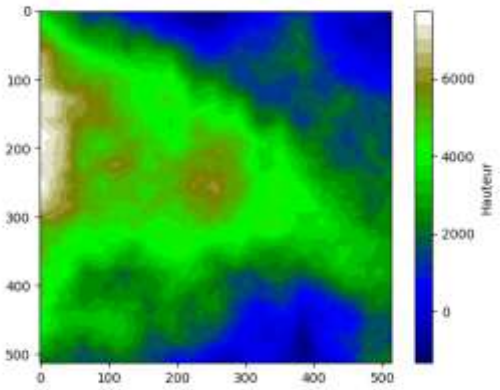
  - Taille de la file = **Variant.**

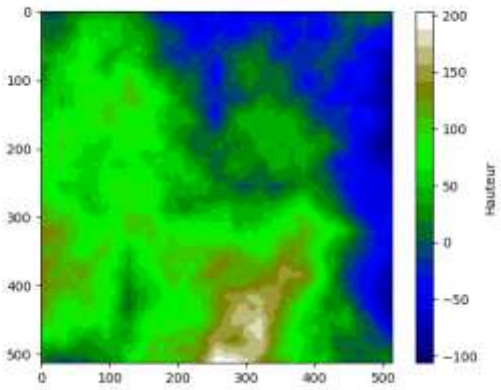
- **Invariant de boucle:**

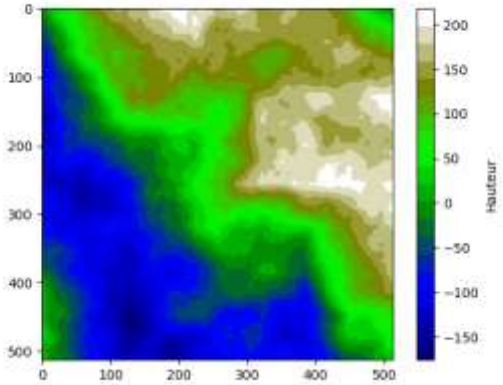
  (P): $Fille\ est\ vide$, permet la **terminaison.**

*La condition que les sommets des carrées prochains sont entiers: $\lfloor droit - gauche \rfloor \geq 1,\ et\ \lfloor haut - bas \rfloor \geq 1$

- Permet de terminer l'ajout des carrées. (Taille de la pile: Bornée)







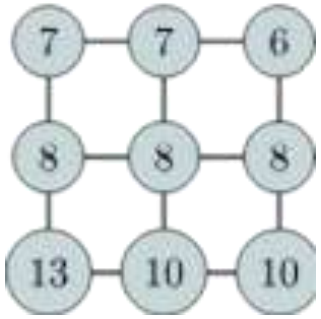
Résultats

On a choisie une palette de couleur pour mieux contraster les constituants du terrain (océans, montagnes, collines, …)

Complexité

Temporelle: $O(n^2)$

- Où n est la largeur de la carte de hauteur (Carré). $n \in 2^{\mathbb{N}} + 1$
- Justifications:
  $(n-1)^2$ est le nbr max de carrées dont les sommets sont entiers, donc la boucle « **while** » ferra $\sim(n-1)^2$ ittérations.



n=3, 4 carrés

Spatiale: $O(n^2)$

- La pile contiendra au max $(n-1)^2$ carrées
- Le résultats est un tableur bidimensionnel de taille n x n

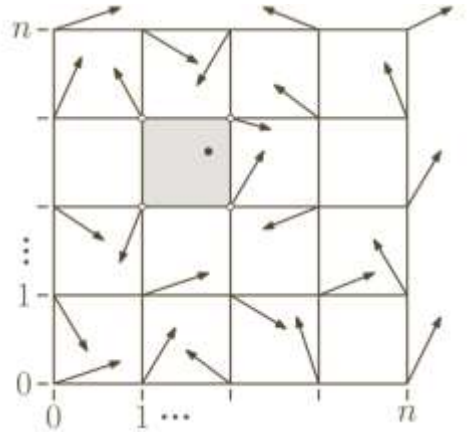
Déplacement du point médian: Complexité

18 Bruit de perlin

- $Perlin2d : \mathbb{R}^2 \rightarrow \mathbb{R}$
- Génère une surface: terrain virtuelle.

1) **Approche**
2) **Résultas**
3) **Complexié**

Etape 1/3: Ingrédients

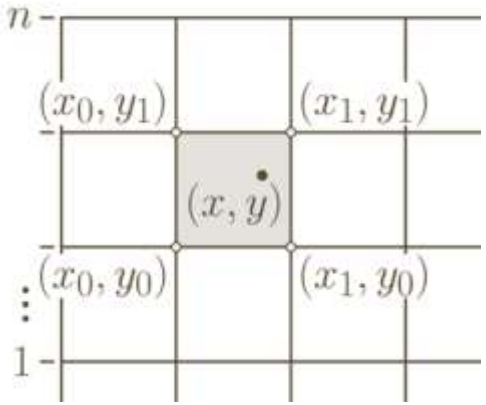


- Subdiviser l'espace en une grille.
- Vecteurs gradients: $g[i,j]$, $(i,j) \in \{0,1,\dots,n\}^2$
  - Unitaires
  - Orientation aléatoires

Pour une des pixels (x,y) où on veut déterminer la hauteur:

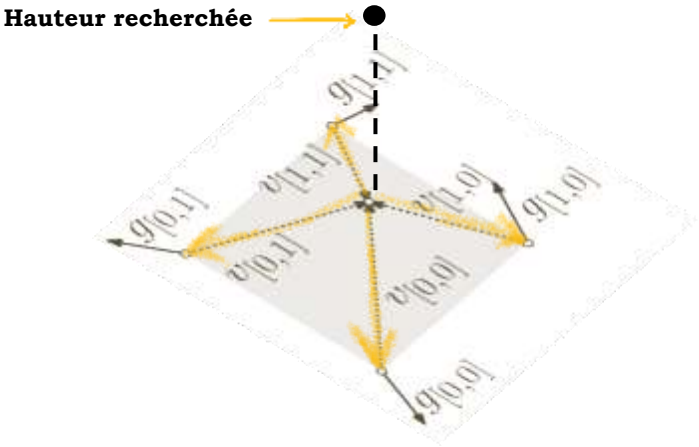
- Elle ∈ à une cellule dont les sommets sont

$$(x_i, y_j), \qquad (i,j) \in \{0,1\}^2$$



- Enfin, on définie les 4 vecteurs distances par:

$v[i,j] = (x,y) - (x_i, y_j)$; $(i,j) \in \{0,1\}^2$



- **Bilan Etape 1/3:**

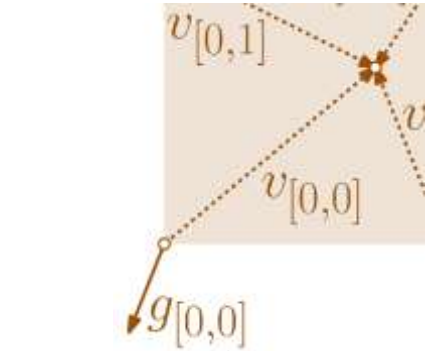
1. Une Grille
2. Vecteurs gradients unitaires aléatoires : $g[i,j]$, $(i,j) \in \{0,1,\dots,n\}^2$
3. 4x Vecteurs distances pour chaque pixel : $v[i,j] = (x,y) - (x_i, y_j)$; $(i,j) \in \{0,1\}^2$

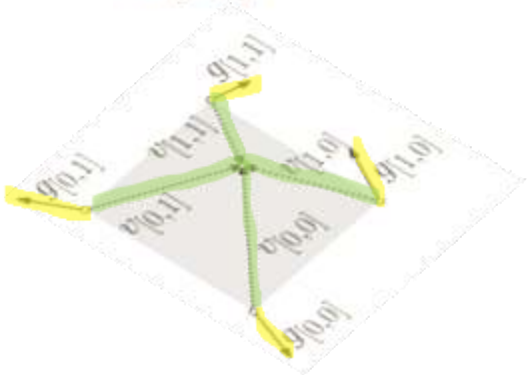
Etape 2/3: Produits scalaires !

Soit $(i,j) \in \{0,1\}^2$:

Sens physiques:

- g[i,j] : direction de la pente local du terrain.
- v[i,j] : sens de déplacement du joueur sur terrain.
- Si v[i,j] et g[i,j] ont le même sens: On monte ↗. (+)
- Sinon (sens opposé) : On descend ↘ . (-)
- Si v[i,j] ⊥ g[i,j]: altitude constante. (0)



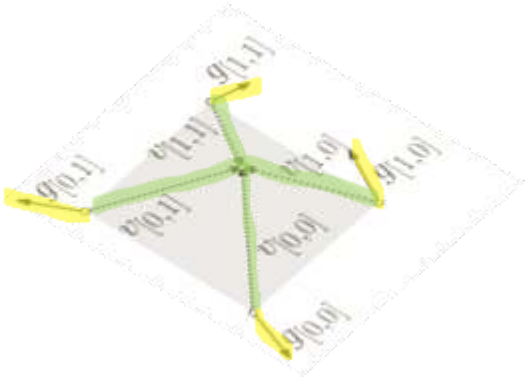


Le produit scalaire produit précisément ce genre de comportement.

- Définissons pour $(i,j) \in \{0,1\}^2$:

$$\delta[i,j] = (v[i,j] \mid g[i,j])$$

Chaque P. Scalaire -> pente de terrain en (x, y)



- **Bilan Etape 2/3:**

1. Calcul de 4 produits scalaires: $\delta[i,j] = (v[i,j] \mid g[i,j])$
2. Chaque scalaire informe sur la pente du terrain local en (x, y).

Comment savoir valeur final de la pente ? -> (3/3)

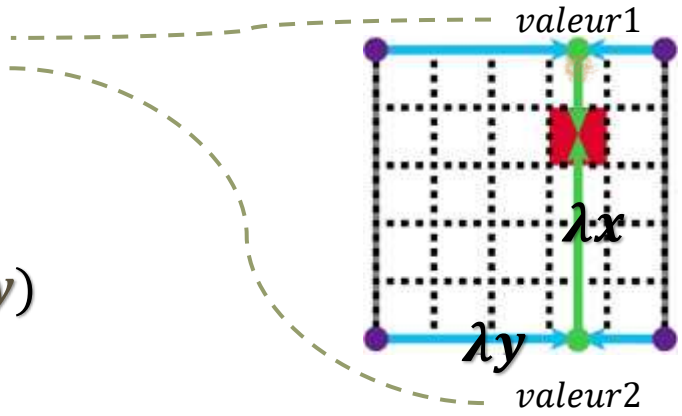
Etape 3/3: interpolation

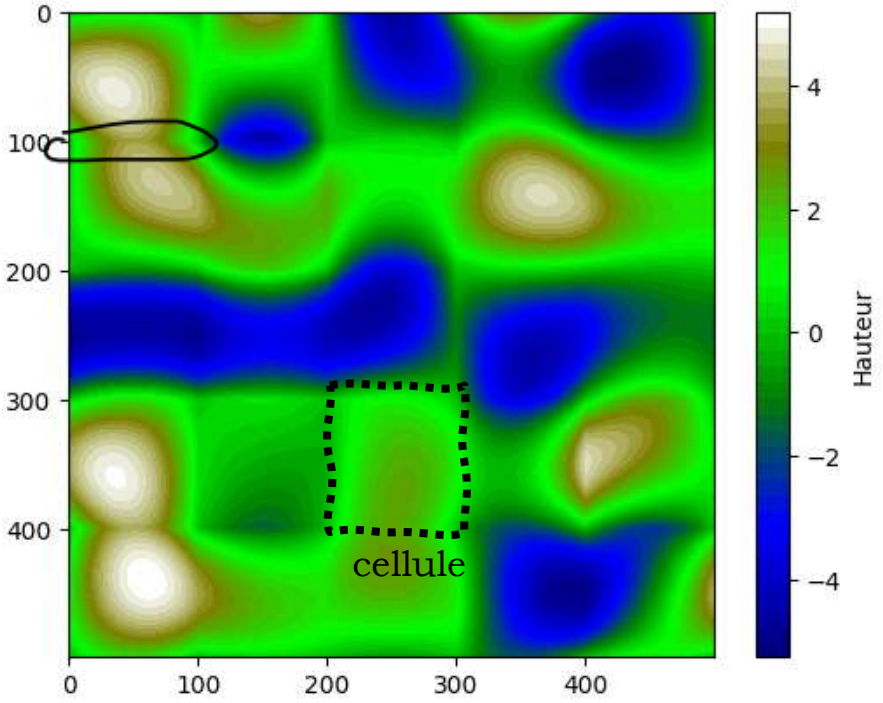
- On a 4 valeurs à combiner:
  - Interpolations linéaires successives entre les $\delta[i,j]$, avec les poids:

$$\lambda x = x - \lfloor x \rfloor, \quad \textbf{et} \quad \lambda y = y - \lfloor y \rfloor \quad \textbf{(Les poids d'interpolation)}$$

$$valeur1 = \delta[0,0]\,.\,(\mathbf{1} - \lambda x) + \delta[0,1].(\lambda x)$$
$$valeur2 = \delta[1,0]\,.\,(\mathbf{1} - \lambda x) + \delta[1,1].(\lambda x)$$

$$resultat(x,y) = valeur1.(\mathbf{1} - \lambda y) + valeur2.(\lambda y)$$





Résultat

Problème:

- On remarque des déformations aux bords de chaque cellule !
- Discontinuité de la fonction perlin2d sur les bords des cellules !

Interpretation:

- Les vecteurs gradients g[i,j] changent d'une cellule à une autre.
- D'où la discontinuité.

Problème des Bords dans le Bruit de Perlin !

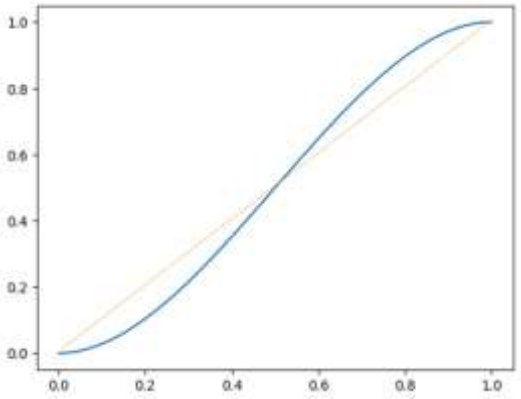
- La fonction ***smoothstep*** permet de créer des transitions douces et progressives entre les points.

- Dans l'algorithme de bruit de Perlin, nous appliquons ***smoothstep*** aux poids d'interpolation des vecteurs gradients pour lisser les transitions aux bords.

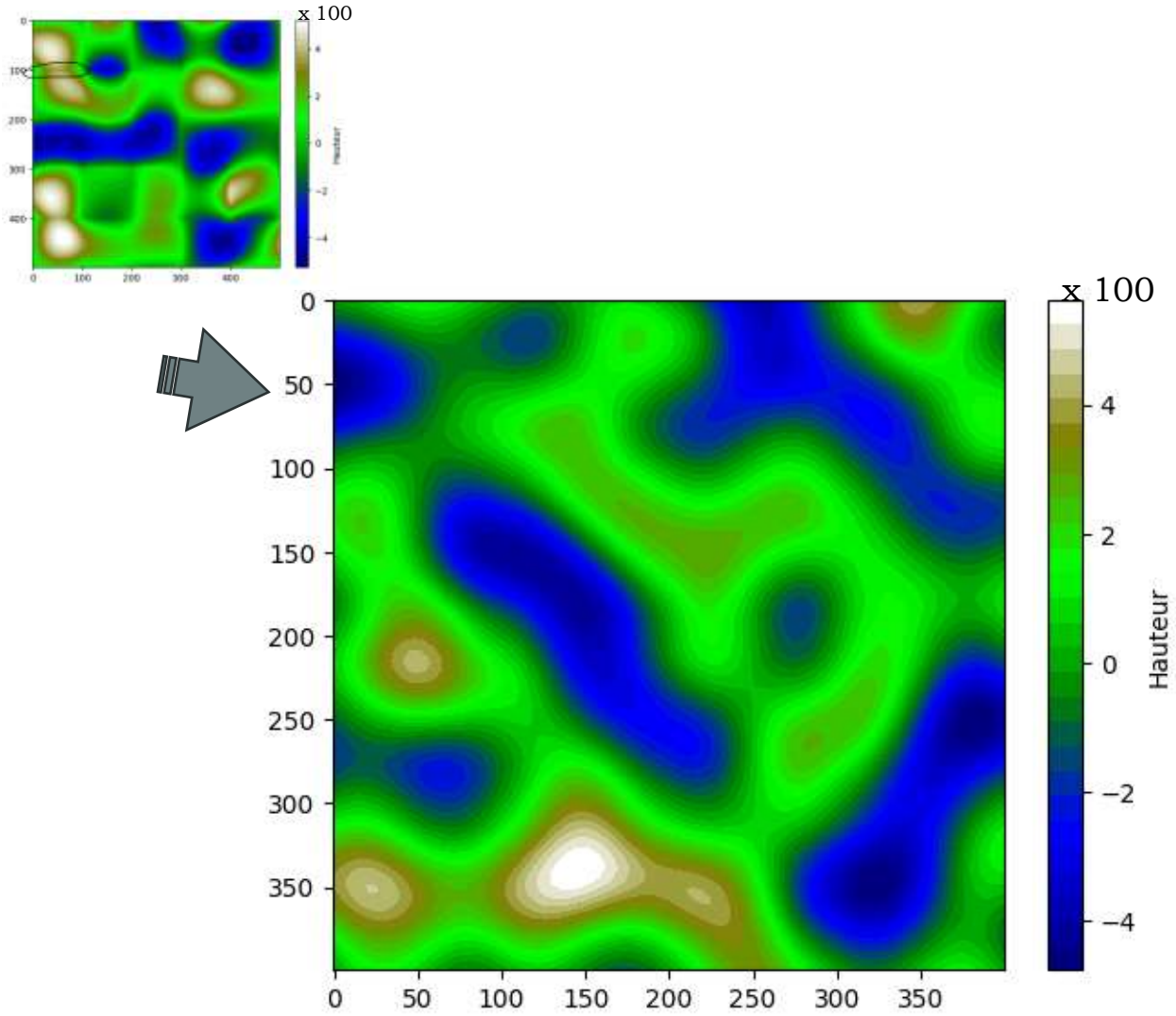
- Et donc accentuer l'effet *des vecteurs gradient les plus proches des côtés:*

$$\lambda x = smoothstep(x - \lfloor x \rfloor),$$
$$\lambda y = smoothstep(y - \lfloor y \rfloor)$$

$$smoothstep(x) = \begin{cases} 0, & x \leq 0 \\ 3x^2 - 2x^3, & 0 \leq x \leq 1 \\ 1, & 1 \leq x \end{cases}$$



Cette fonction interpole entre 0 et 1 de manière lisse, évitant ainsi les changements abrupts.



Bruit de perlin

- Vue Aérienne

Bruit de perlin 2D: Résultats

De plus:

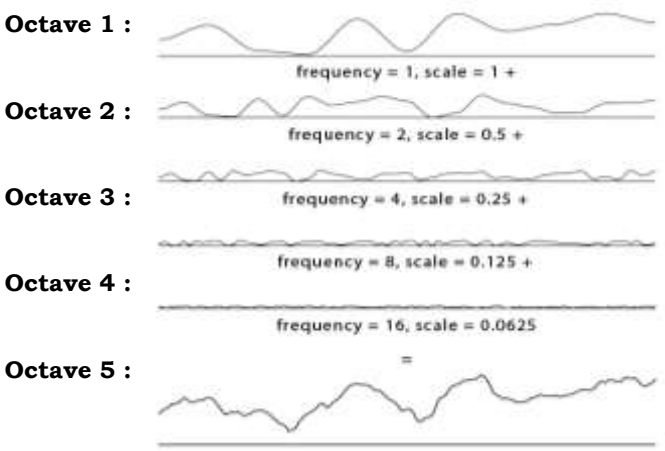
- Pour plus de détailles:
- Additionner plusieurs cartes de Perlin2d (**octaves**), avec des paramètres variées.

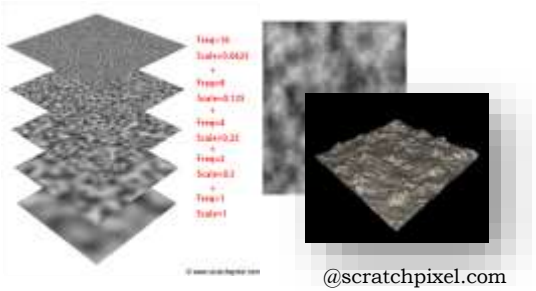
✓ Exemple:

D'un octave à l'autre:

- La fréquence se multiplie par 2, 2 : c'est la **lacunarité**.
- L'amplitude se multiplie par 1/ 2, ½ : c'est la **persistance**

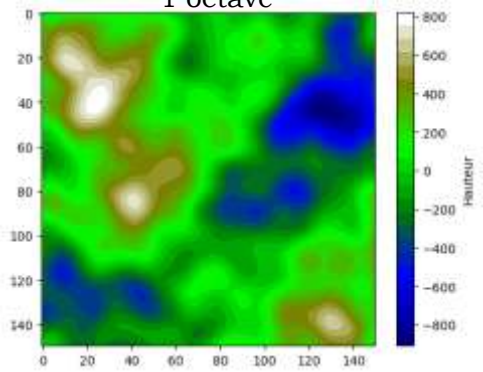
2D



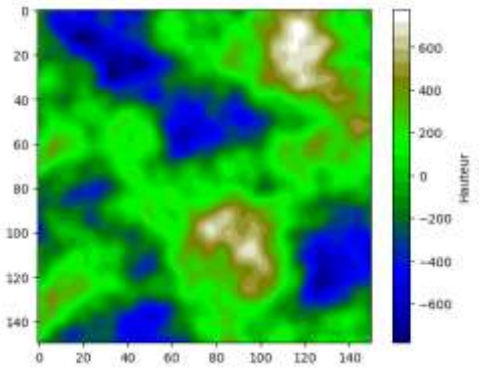


3D

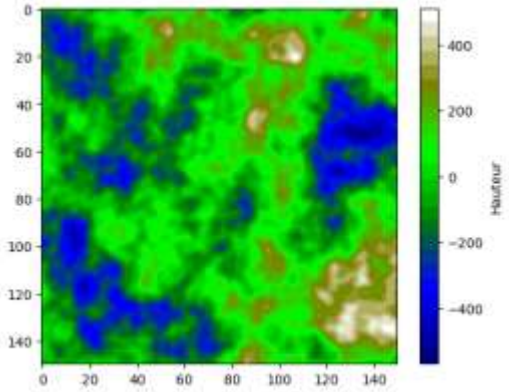
1 octave



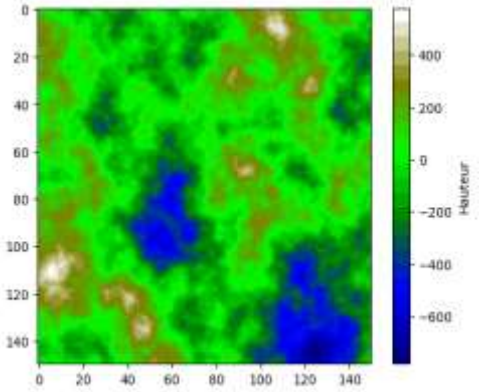
2 octaves



4 octaves



5 octaves



Bruit de perlin

Persistance: 2

Lacunarité: 1/2

Complexité

- Temporelle:
  - Perlin2d (additions, multiplications, …): $O(1)$
  - Si on a un tableau $n \times n$, on appel Perlin2D n^2 fois.
  - ➔ $Complexité\ finale$: $O(n^2)$ quadratique !
- Spatiale:
  - Le résultats est un tableur bidimensionnel de taille n x n : $O(n^2)$

30 Valeurs ajoutées

1) Visualisation 3D
2) Test des performances et confrontation
3) Validation

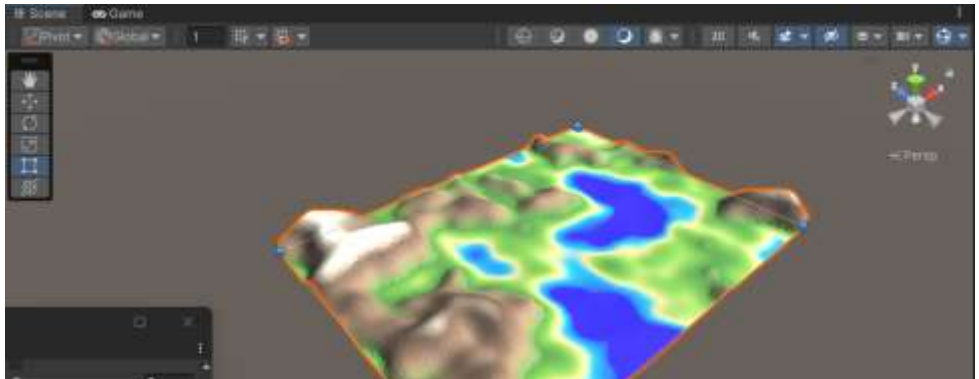


fig: Simulation 3d du bruit de perlin

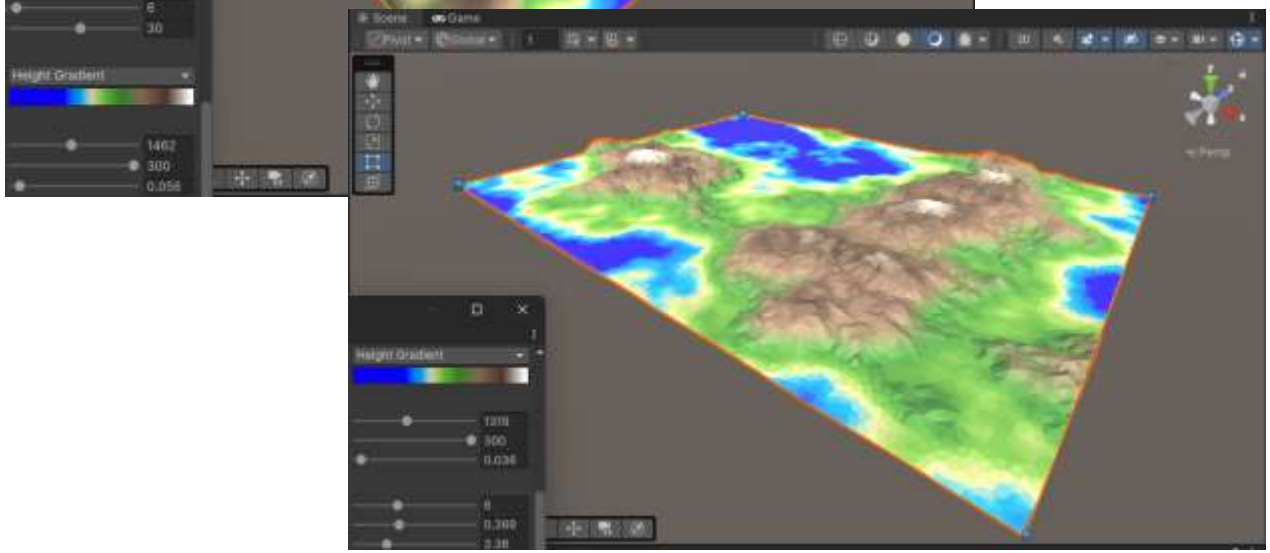


fig: Simulation 3d du MDP

Visualisation 3d

Algorithmes: MPD/Bruit de perlin + nuances colorées

Logiciel: Unity 6.0

Environnement de simulation:

Critère	Niveau
Processeur	AMD Ryzen 5 5600U, 6 cœurs, 2.3Ghz
Carte Graphique	AMD Radeon Vega 7
RAM	8 GB 2666 MHz
Moteur de Jeu (Logiciel):	Unity Engine
Environnement de Test	Pas d'autres processus lourds sur le système pendant les tests.

- **Remarquez** le caractères ordinaire de la machine.
  Vision: Grand public.

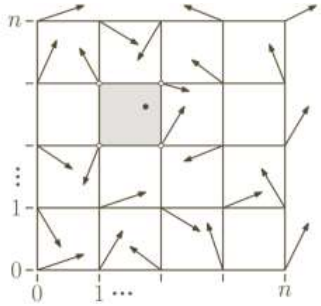
Définition des critères de performances

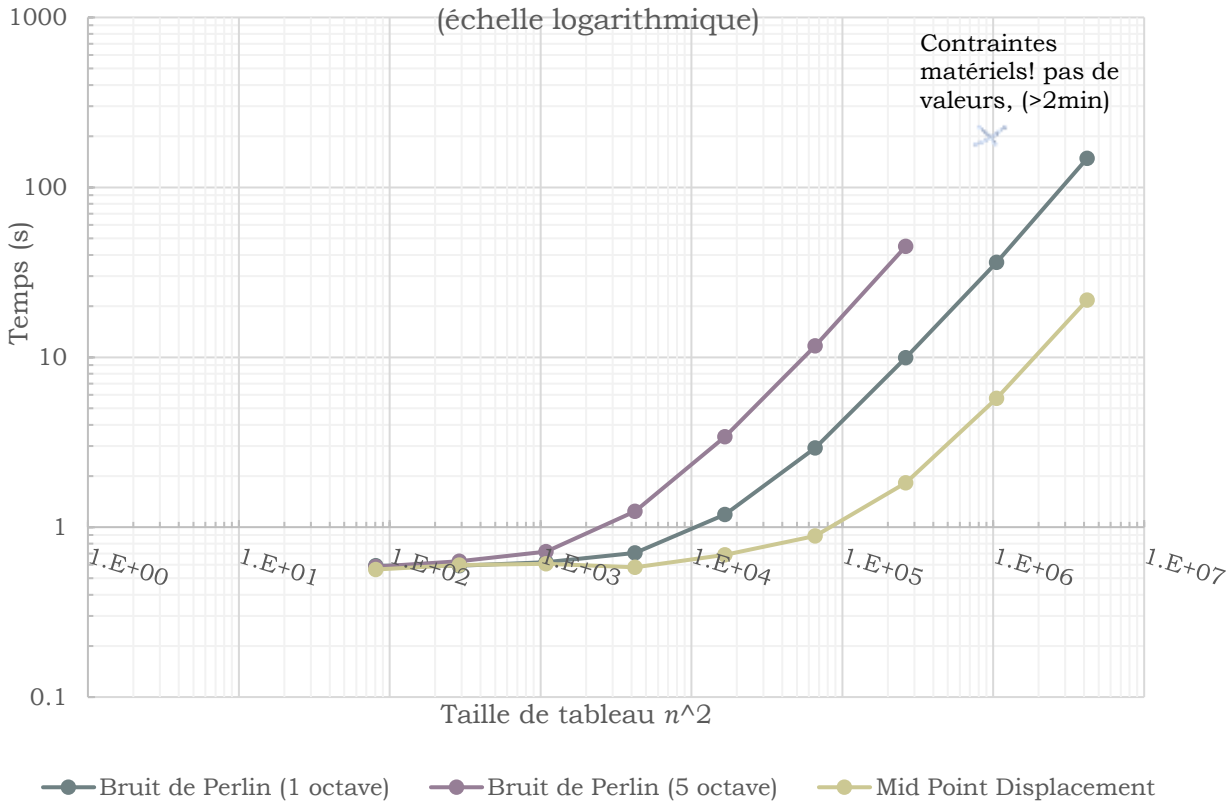
Pour répondre à notre problématique:
On choisira 3 critères parmi les critères d'évaluation suivant:

Catégorie	Métrique	Description	Critère choisie dans le TIPE
Réalisme	Nombre de pics et de vallées	Calculer le nombre de points hauts et bas pour la diversité topographique	
	Fractalité	Mesurer la complexité et les détails à différentes échelles	
	Réseau de drainage	Vérifier le réalisme des rivières et lacs simulés	
	Erosion	Vérifier si les effets d'érosion sont réalistes	
Performance	Temps de calcul	Temps nécessaire pour générer un terrain	**x**
	Utilisation de la mémoire	Quantité de mémoire utilisée pendant la génération	**x**
	Taille du terrain	Tester l'algorithme sur différentes tailles de terrain	**x**
	Parallélisation	Impact de la parallélisation sur la performance	
	Résistance aux paramètres extrêmes	Comportement de l'algorithme avec des valeurs de paramètres extrêmes	
	Consistance	Variation des résultats avec les mêmes paramètres d'entrée	**x**
Jouabilité	Temps de chargement	Temps nécessaire pour charger le terrain dans un moteur de jeu	
	Feedback des testeurs	Retours d'utilisateurs sur la jouabilité et le réalisme perçu	

Valeurs ajoutées: Test de performances

Temps D'exécution





Commentaire:

Le MPD est plus rapide à calculer que le bruit de Perlin. Car le MPD peut être calculé en une seule passe, tandis que le bruit de Perlin nécessite plusieurs passes pour générer le même niveau de détail.

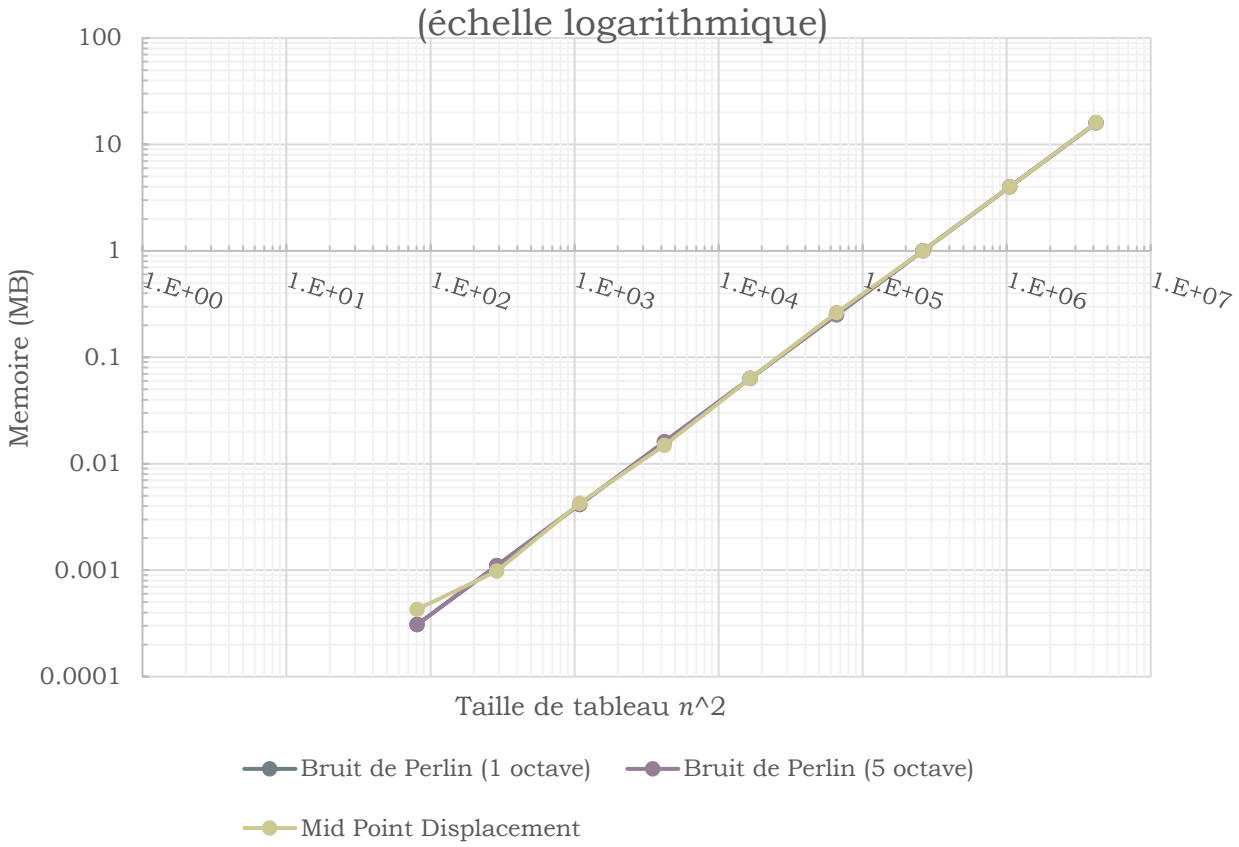
Module:

timeit

- 10 répétitions par test
- Incertitude de type A

Valeurs ajoutées: Test de performances

Mémoire occupée



- Complexité: spatiale: $O(n^2)$
- Tout les algorithmes produisent au final un tableau de valeur de taille $\boldsymbol{n^2}$
- Chaque valeur: contient (hauteur + code couleur) (~ **4 Octets**)
- Mémoire occupée:

$$\frac{4n^2}{1024^2}\ MB$$

Valeurs ajoutées: Test de performances

Confrontation

Critère	Bruit de Perlin	Déplacement du point médian
Type	Gradient (dégradé de valeurs)	Fractale récursif
Lissage	Terrain lisses et continus	Terrain rugueux et semblable aux fractales
Contrôle sur les détails *	Fréquences, octaves, amplitudes ajustables.	Profondeur de récursion, l'amplitude de perturbation
Utilisations	Textures, terrains, nuages, motifs naturels	Terrains, paternes brisés (Feuilles)
Dépendances	Nécessite un gradient de valeurs aléatoires prédéfinies	Commence par 4 valeurs

* Obtenir des caractéristiques très spécifiques peut être difficile.
Des post-traitements complexes sont utiles (dont on va pas traiter !)

Validations

- Utiliser le MDP pour des applications nécessitant une génération rapide et efficace.
- Intégrer le Bruit de Perlin pour ajouter des détails et améliorer le réalisme des terrains.
- Évaluer les besoins spécifiques du jeu (réalisme vs performance) pour choisir la meilleure approche ou combinaison des deux.

- En conclusion, la génération procédurale des terrains, lorsqu'elle est bien implémentée et optimisée, peut considérablement enrichir l'expérience de jeu tout en respectant les contraintes de performance et des ressources.

```python
def carte_mid_point(taille, alea_in = 0):
    #initialisation de la carte de hauteur
    # avec les quatres sommets du carrée initial
    res = [[0]*taille for i in range(taille)]

    res[0][0] = random.randint(-alea_in, alea_in)
    res[0][taille-1] = random.randint(-alea_in, alea_in)
    res[taille-1][0] = random.randint(-alea_in, alea_in)
    res[taille-1][taille-1] = random.randint(-alea_in, alea_in)

    #Initialisation de la file
    file = deque()
    file.append((0, 0, taille-1, taille-1, alea_in))

    while(file):
        #Calcul des coord des médians de chaque coté et du centre
        xg, yg, xd, yd, alea = file.popleft()
        xc, yc = (xg+xd)//2,  (yg+yd)//2

        # Affectation des hauteurs aux points médians et du centre
        res[xc][yc] = (res[xg][yg] + res[xg][yd] \
                       + res[xd][yd] + res[xd][yg]) // 4 + random.randint(-alea, alea)
        res[xc][yg] = (res[xg][yg] + res[xd][yg]) // 2 + random.randint(-alea, alea)
        res[xc][yd] = (res[xg][yd] + res[xd][yd]) // 2 + random.randint(-alea, alea)
        res[xg][yc] = (res[xg][yg] + res[xg][yd]) // 2 + random.randint(-alea, alea)
        res[xd][yc] = (res[xd][yd] + res[xd][yg]) // 2 + random.randint(-alea, alea)

        #Ajout des carrées decouverts à la file
        if xc - xg > 1:
            file.append((xg, yg, xc, yc, alea//2))
            file.append((xc, yc, xd, yd, alea//2))
            file.append((xg, yc, xc, yd, alea//2))
            file.append((xc, yg, xd, yc, alea//2))

    return res
```

ANNEXE

Implémentation du

« Mid Point Displacement ».

```python
def perlin_2d(x, y, grad):
    #Coordonnées du sommet inf-gauche de la cellule actuelle
    X, Y = int(x), int(y)

    #coordonnées relatives à la cellule actuelle
    dx = x-X
    dy = y-Y

    #Produits scalaires:
    ll = dx * np.cos(grad[X,Y]) + dy * np.sin(grad[X,Y])
    lr = (dx-1) * np.cos(grad[X+1,Y]) + dy * np.sin(grad[X+1,Y])
    ul = dx * np.cos(grad[X,Y+1]) + (dy-1) * np.sin(grad[X,Y+1])
    ur = (dx-1) * np.cos(grad[X+1,Y+1]) + (dy-1) * np.sin(grad[X+1,Y+1])

    #Prioriser le poids des sommets proches par smooth_step
    dx = smooth_step(dx)
    dy = smooth_step(dy)

    #Interpolation et Combinaison des valeurs
    n1 = lerp(dx, ll, lr)
    n2 = lerp(dx, ul, ur)
    return lerp(dy, n1, n2)
```

ANNEXe

Implémentation du

« bruit de perlin ».

```
# Génération de la carte de hauteur
res = carte_mid_point(513, 200)
data = np.array(res)

# Création d'une palette couleur personnalisé
colors = [
    (0, 0, 0.5),      # Ocean - bleu
    (0, 0, 1),        # Ocean - blue claire
    (0, 0.5, 0),      # Collines - vert sombre
    (0, 1, 0),        # plateury - vert
    (0.5, 0.5, 0),    # Vallés - Marron
    (1, 1, 1)         # Mountagnes - Blanc
]

cmap = mcolors.LinearSegmentedColormap.from_list('terrain_map', colors, N=25)

# Visualisation de la carte
fig, ax = plt.subplots()
im = ax.imshow(data, cmap=cmap)
cbar = plt.colorbar(im, ax=ax)
cbar.set_label('Hauteur')
plt.show()
```

Génération des cartes 2d, par:

Outils:

- Matplotlib
- Numpy

```python
def carte_bruit_perlin(N, freq, ampl):
    grad = np.random.uniform(0, 2*np.pi, size=(N+1, N+1))
    _map = []
    for i in range(N*freq):
        l  = []
        for j in range(N*freq):
            v = perlin_2d(i/N, j/N, grad) * ampl
            l.append(v)
        _map.append(l)

    _map = np.array(_map)
    return _map
```

ANNEXE

Implémentation du bruit de perlin afin de générer toute une carte de hauteur.

```python
tableau_hauteurs = gen_carte_perlin(20, 5, 10)

# Création d'une palette couleur personnalisé
colors = [
    (0, 0, 0.5),      # Ocean - bleu
    (0, 0, 1),        # Ocean - blue claire
    (0, 0.5, 0),      # Collines - vert sombre
    (0, 1, 0),        # plateury - vert
    (0.5, 0.5, 0),    # Vallés - Marron
    (1, 1, 1)         # Mountagnes - Blanc
]

_cmap = mcolors.LinearSegmentedColormap.from_list('terrain_map', colors, N=15)

fig, ax = plt.subplots()
im = ax.imshow(tableau_hauteurs, cmap=_cmap)
cbar = plt.colorbar(im, ax=ax)
cbar.set_label('Hauteur')

plt.show()
```

Plus d'annexes en version imprimé.

Visualisation de la cartes 2d:

Bruit de perlin

Outils:

- Matplotlib
- Numpy

Votre attention est précieuse, merci 🙏

Lien vers le simulateur que j'ai créer:
https://github.com/K-anwar/TIPE-2024-Jeux_et_sports

OUTILS

1. Moteur de jeu: **Unity 3D**
2. Langages**: Python, C#**
3. Modules Python**: Numpy, Matplotlib, Collections, time.**
4. Modules C#: **UnityEngine, System.Collections, System.Collections.Generic;**

Références

1. The Book of Shaders
2. Understanding Perlin Noise (adrianb.io)
3. The Perlin noise math FAQ (mzucker.github.io)
4. Proceedings Template - WORD (nyu.edu)
5. Unity Documentation
6. lect13-2d-perlin.pdf (umd.edu)
7. https://www.reddit.com/r/Unity3D