Relazione Foundations of Game Design

Innovazioni nella Generazione Procedurale di Terreni

Matteo Breganni 869549

Sommario

Introduzione

Nel presente lavoro sono stati analizzati due articoli riguardanti la generazione procedurale di terreni: *Procedural terrain generation with style transfer (2024)* e *StyleTerrain: Disentangled generative model for controllable high-quality procedural terrain generation (2023)*. Il primo presenta un approccio ibrido che unisce tecniche procedurali classiche, come il *Perlin Noise*, al *Neural Style Transfer* per ottenere terreni stilizzati ispirati a dati reali, mantenendo bassi i requisiti computazionali e aumentando la flessibilità creativa. Il secondo propone un modello basato su una variante di StyleGAN2 per generare heightmap realistiche e modificabili grazie al disentanglement dello spazio latente, ideale per applicazioni dove si richiede una modifica precisa e parametrica del terreno. Entrambi i metodi mostrano soluzioni innovative per migliorare realismo, qualità e personalizzazione nella generazione di paesaggi virtuali.

Introduzione alla generazione procedurale

La generazione procedurale di terreni è una tecnica fondamentale nel campo dei videogiochi e della computer grafica, utilizzata per creare ambienti virtuali immersivi e realistici. Questo processo permette di generare in modo automatico paesaggi come montagne, valli, fiumi e coste, simulando la complessità del mondo naturale senza richiedere un design manuale. Nei videogiochi, questa tecnica consente di costruire mondi vasti e dinamici con un minimo intervento umano, riducendo i costi di produzione e aumentando la varietà delle esperienze di gioco. Tuttavia, uno dei maggiori problemi dei metodi tradizionali è il bilanciamento tra realismo, controllabilità e efficienza computazionale.

Le tecniche di generazione di terreni possono essere divise in 3 categorie:

- metodi procedurali,
- metodi basati su simulazioni,
- metodi basati su esempi (di cui fanno parte gli approcci basati sul machine learning).

Metodi Procedurali

I metodi procedurali si basano su algoritmi che di solito generano heightmap 2D. Questi approcci sono noti per produrre risultati di alta qualità, soprattutto nella creazione di texture dettagliate e realistiche. Tuttavia, la loro natura casuale rende complessa la produzione di terreni con caratteristiche specifiche e controllabili, limitando la possibilità di ottenere risultati precisi e riproducibili.

Metodi Basati su Simulazioni

Gli algoritmi basati sulle simulazioni provano a riprodurre fenomeni naturali per creare terreni più plausibili, con caratteristiche naturali. Tra questi metodi troviamo principalmente metodi basati sull'erosione. I risultati hanno un qualità molto alta, molto simile alla realtà, ma richiedono un'altissima conoscenza dei fenomeni naturali fisici, della geologia... per poter prevedere e controllare i risultati.

Inoltre, questi metodi possono essere molto dispendiosi a livello computazionale, più livelli di dettaglio vengono aggiunti.

Metodi Basati su Esempi

I metodi basati sugli esempi partono dalla costruzione di un dataset di dati reali. Durante la generazione, diversi frammenti sono uniti per creare un nuovo terreno.

Nonostante questi metodi possano creare risultati di alta qualità, data la loro necessità di dati, questo tipo di approccio è limitato dalla qualità dei dati e dalla loro distribuzione, non sono in grado di riprodurre caratteristiche non presenti all'interno del dataset.

I metodi basati su approcci di machine learning sono una sotto-parte di questa categoria.

Approcci Basati su Machine Learning

Gli approcci basati su machine learning si distinguono per la capacità di apprendere direttamente da un dataset di terreni reali, senza la necessità di definire manualmente le regole di generazione.

Le Generative Adversarial Networks (GANs), sono usate per analizzare e replicare le caratteristiche strutturali e stilistiche presenti nei dati forniti, data la loro capacità di apprendere pattern complessi, permettendo di generare terreni altamente realistici e diversificati. Tuttavia, anche questi metodi sono vincolati dalla qualità e dalla varietà del dataset utilizzato, poiché non possono produrre configurazioni che non facciano parte della distribuzione dei dati di training.

Generazione Procedurale di Terreno con Style Transfer

Questo metodo si distingue per una maggiore versatilità, requisiti hardware ridotti e un'integrazione più efficace nel processo creativo da parte dei designer rispetto a semplici approcci procedurali, alle GAN, al design manuale o all'interpolazione di dataset geospaziali.

La tecnica utilizza la generazione di noise maps attraverso un algoritmo multi-layered Gaussian noise o, in alternativa, l'algoritmo di Perlin noise. Successivamente, viene applicata una trasformazione neurale basata sul Neural Style Transfer, che permette di trasferire lo stile da heightmap reali a quelle generate proceduralmente.

Unendo la potenza della generazione algoritmica con il processing neurale, questo approccio ha il potenziale di produrre terreni altamente realistici e coerenti dal punto di vista morfologico, mantenendo al contempo un alto grado di personalizzazione.

I metodi tradizionali (manual design e interpolazione di dataset geospaziali) hanno problemi di scalabilità, adattabilità e risorse computazionali necessarie. Metodi come il Perling noise vengono utilizzati dato che occupano poco spazio su disco (dato che bisogna solamente salvare le regole di generazione) e permettono una generazione randomica.

Un interessante use case di questo metodo è la creazione di ambienti virtuali per videogiochi e simulazioni, dove serve una grande varietà di paesaggi realistici e personalizzabili. Ad esempio, un game designer può generare una base morfologicamente plausibile con il Perlin Noise, come un sistema montuoso o una pianura ondulata, e poi applicare uno stile derivato da una heightmap reale, ad esempio delle Alpi o dell'Himalaya. Questo permette di ottenere risultati molto realistici senza dover modificare manualmente ogni dettaglio o dipendere da grandi dataset geografici, tutto con un ridotto costo computazionale.

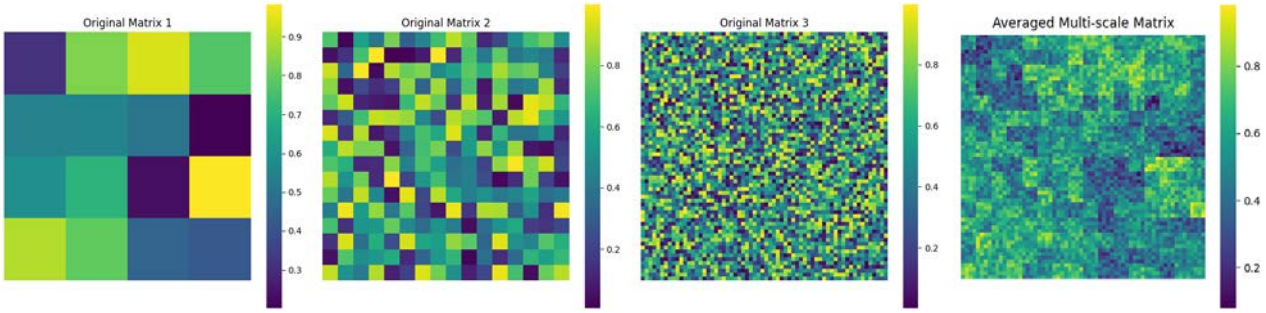
Generazione Procedurale

Nel paper di riferimento, la fase di generazione procedurale è applicata secondo un algoritmo multi-layered Gaussian noise oppure l'algoritmo di Perlin noise.

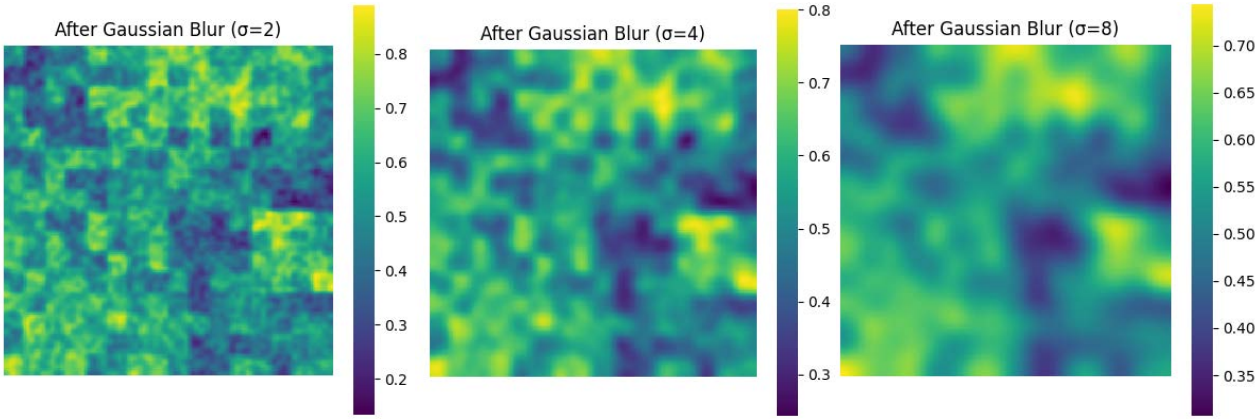
Explicit Noise

Questo metodo genera matrici di dimensioni diverse, composte da numeri casuali distribuiti uniformemente (ovvero dove ciascun valore ha la stessa probabilità di essere un valore qualsiasi nel range predefinito). Successivamente, viene applicato un processo di upscale per ridimensionare le matrici alla dimensione finale desiderata e infine si calcola la media delle matrici per ottenere la mappa di rumore finale.

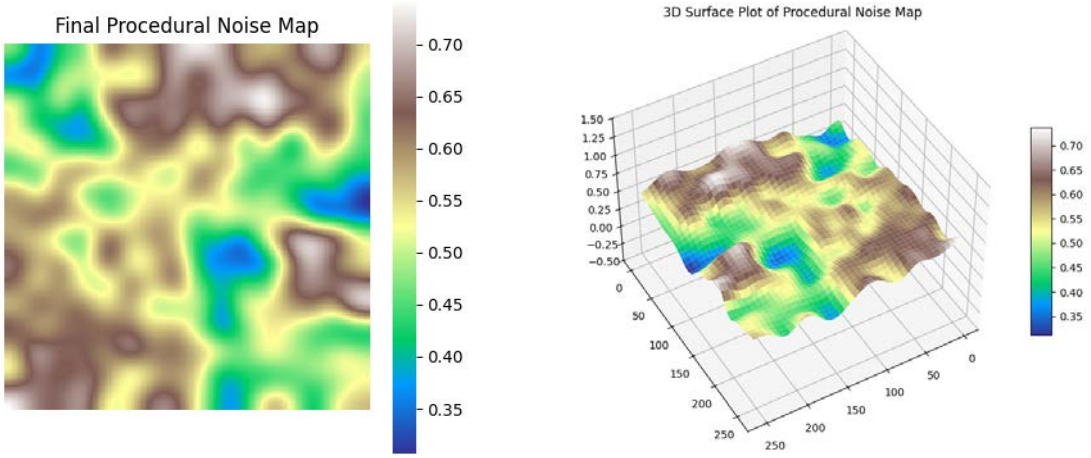
Il fatto che le matrici sono inizialmente di dimensione diversa fa in modo che dopo la media, il rumore non sarà più uniformemente distribuito nella matrice, ma ci saranno aree di alta e bassa densità.



A questo punto vengono applicate delle convoluzioni tramite filtri gaussiani di dimensioni diverse per applicare uno smoothing progressivo.



La heightmap generata è la seguente:



Pur essendo semplice e leggero, questo metodo offre un realismo limitato, infatti non cattura le strutture naturali e le variazioni complesse dei paesaggi reali. Inoltre, lo smoothing gaussiano genera transizioni morbide ma poco strutturate.

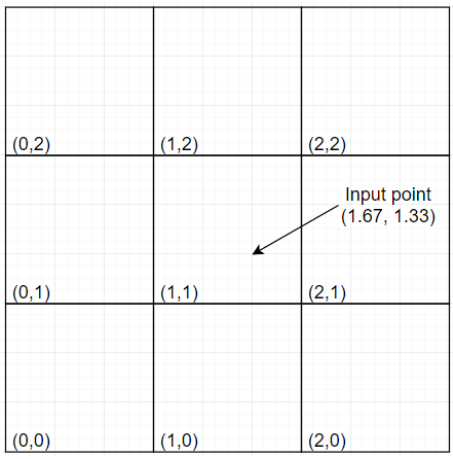
Perlin Noise

Il perlin noise produce risultati che di solito hanno un aspetto più naturale. Uno dei grandi vantaggi di questo metodo è che diverse parti della noise map possono essere calcolate in momenti diversi, senza dover avere calcolato le altre parti intorno. Non è quindi necessario pre-computare l'intera noise map, ma lo si può fare quando è necessario, tramite un sistema a chunk.

A differenza del rumore casuale standard, il Perlin Noise produce valori pseudo-casuali distribuiti in modo coerente, creando risultati che appaiono naturali e realistici. Questa caratteristica lo rende

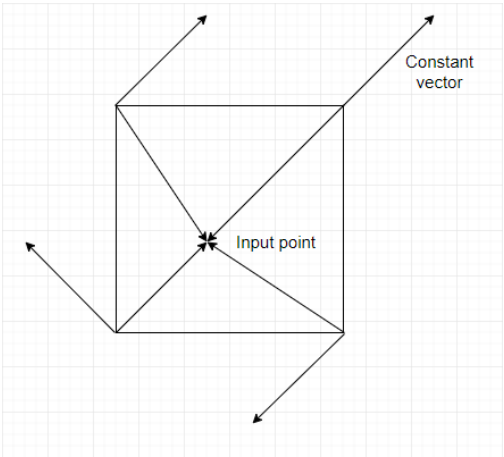
ideale per generare terreni, nuvole, tessuti... qualsiasi superficie che richieda un aspetto organico variabile.

Innanzitutto, viene generata una griglia di punti.



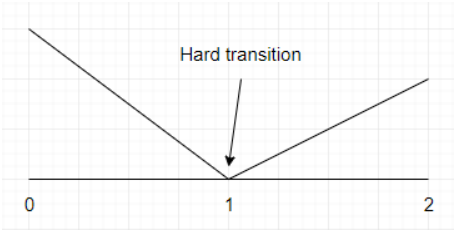
Per calcolare il valore di un punto qualsiasi della heightmap, l'algoritmo guarda ai 4 punti della griglia più vicini. Ad ognuno di questi punti della griglia è stato preassegnato vettore casuale che indica una direzione. L'algoritmo calcola quanto il punto in questione "sente" i vettori al suo intorno, ovvero quanto ciascuno contribuisce al suo valore.

Per fare questo calcolo, l'algoritmo utilizza sia il vettore casuale assegnato a ciascun punto della griglia, che i vettori che partono da ciascuno dei punti della griglia e puntano verso il punto (x, y) per cui si vuole calcolare il Perlin noise.

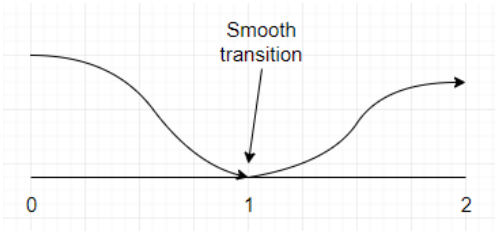


Questo processo viene ripetuto per ogni punto della heightmap, creando un'immagine o una superficie apparentemente casuale ma con coerenza, senza salti improvvisi tra valori adiacenti.

Durante il calcolo di ciascun punto, i vari vettori vengono interpolati. Se venisse usata l'interpolazione lineare, ci sarebbero dei punti di cambiamento bruschi, che non sono naturali:

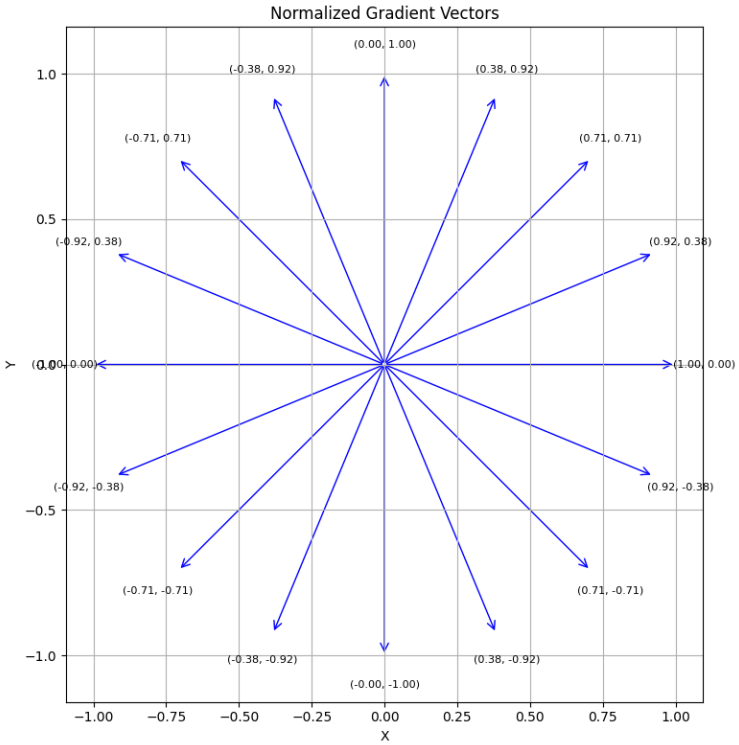


Mentre invece facendo un'interpolazione non lineare, utilizzando questi vettori, abbiamo delle transizioni morbide:



Dettagli Implementativi

A ciascun punto della griglia viene associato un vettore che punta ad una direzione. Spesso, il numero di direzioni viene ridotto per semplicità computazionale. Questi gradienti sono predefiniti, nella mia implementazioni ho precalcolato 16 vettori 2D normalizzati equidistanti:



Le implementazioni possono essere diverse, per esempio si potrebbero anche usare solamente 4 vettori per semplicità, oppure si potrebbero usare 16 vettori 3D come nel paper originale del Perlin Noise, ma questo non è necessario dato che per i terreni è sufficiente generare una heightmap bidimensionale. Vettori equidistanti 3D sarebbero utili nello use case della generazione di un volume.

Per assegnare a ciascun punto della griglia un vettore casuale, viene creata una permutation table di 256 valori, e viene ripetuta 2 volte per comodità:

```
# Hashing permutation table (pseudo-random indexing)
permutation = list(range(256))
np.random.shuffle(permutation)
permutation = permutation * 2  # Extend to allow easier wrapping
```

La lista permutation conterrà quindi tutti i valori tra 0 e 255 mischiati, ed è ripetuta 2 volte.

Ciascun punto della griglia sarà quindi in una posizione (x, y). Per recuperare il valore corretto dalla tabella di permutazione, i valori di x e y vengono convertiti nel range 0-255 di modo che rimangano nella lunghezza della tabella di permutazione (viene usata una bitmask per velocizzare il conto):

```
X = int(np.floor(x)) & 255
Y = int(np.floor(y)) & 255
```

I valori dei 4 punti della griglia intorno al punto (x, y) del quale si vuole calcolare la Perlin noise sono quindi recuperati dalla tabella di permutazione tramite:

```
aa = permutation[permutation[X] + Y]
ab = permutation[permutation[X] + Y + 1]
ba = permutation[permutation[X + 1] + Y]
bb = permutation[permutation[X + 1] + Y + 1]
```

Dato che X e Y possono essere valori fino a 255, questa somma interna alle parentesi quadre potrebbe produrre un indice più alto di 255, e per questo motivo la tabella di permutazione è ripetuta 2 volte. Questo metodo permette di recuperare per ogni punto della griglia (x, y) un valore randomico ma costante, tramite la tabella di permutazione. Il valore verrà poi utilizzato per recuperare uno dei 16 gradienti predefiniti, facendo un'operazione di modulo (*valore % 16*), utilizzando una bitmask anche qui per velocizzare il calcolo (*valore & 15*).

Per calcolare il valore di Perlin Noise nel punto, vengono "normalizzate" le coordinate (x, y) del punto all'interno del quadrante in cui si ritorvano:

```
x -= np.floor(x)
y -= np.floor(y)
```

quindi se x=3.51, dopo l'operazione sarà x=0.51, che è una misura di posizione relativa all'interno del quadrante composto dai 4 punti della griglia al cui è all'interno.

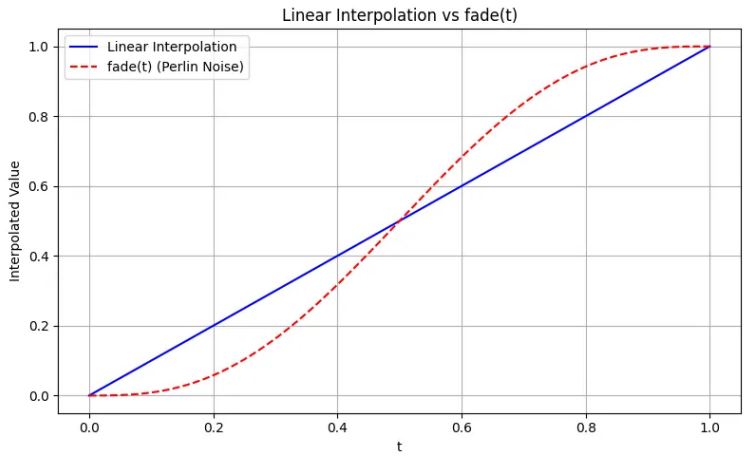
Questo 2 valori vengono poi modificati singolarmente tramite questa funzione:

```
def fade(t):
    """Smooth fading function for interpolation."""
    return t * t * t * (t * (t * 6 - 15) + 10)
```

```
u = fade(x)
v = fade(y)
```

l'operazione di fade è usata per interpolare tra i grid points in modo graduale.

La funzione applica quindi una interpolazione non lineare tramite questo polinomio di quinto grado, per evitare transizioni brusche come nell'interpolazione lineare. Il polinomio è stato definito nel paper di *Ken Perlin "Improving Noise"*, sostituendo quello proposto precedentemente, per risolvere degli artefatti visivi causati dalla discontinuità della derivata seconda agli estremi della funzione.



Questi nuovi valori *u* e *v* verranno usati come pesi (parametro *t*) nella funzione di interpolazione, per renderla graduale e quindi non lineare:

```
def lerp(t, a, b):
    """Linear interpolation."""
    return a + t * (b - a)
```

questa funzione applica un'interpolazione lineare tra a e b, dato un valore t compreso tra 0 e 1 che rappresenta quanto il punto sia vicino ad a oppure b.

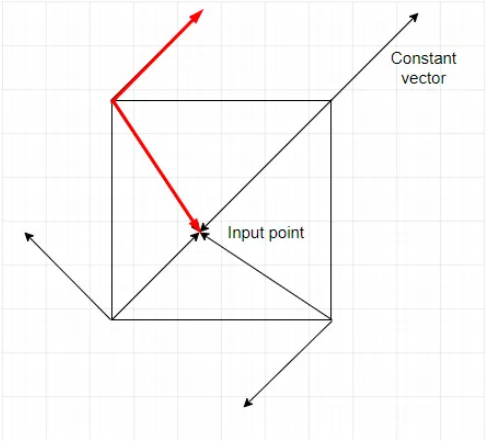
I valori a e b tra cui interpolare sono però da calcolare, dato che dipendono dai 4 vettori direzionali assegnati ai 4 punti della griglia intorno al punto di cui stiamo calcolando il Perlin noise.

Il valore che era stato recuperato dalla tabella di permutazione viene passato come *hash*:

```
def grad(hash, x, y):
    h = hash & 15  # 4 bit → 16 possible values
    g = gradients_16[h]
    return x * g[0] + y * g[1]
```

questa funzione calcola il prodotto scalare (dot product) tra il vettore del punto della cella (uno dei 4 punti che circondano il punto per cui stiamo calcolando il Perlin noise) e il vettore (x, y), ovvero il vettore distanza (offset vector) dal punto della griglia al punto su cui stiamo valutando il Perlin noise.

Il dot product è quindi calcolato per esempio tra questi 2 vettori:



Il risultato serve per capire quanto il gradiente "spinga" verso il punto: più i due vettori sono allineati e maggiore sarà il valore.

Le contribuzioni del punto in alto a sinistra (*aa*) e del punto in alto a destra (*ba*) vengono quindi interpolate in *x1*, mentre le contribuzioni del punto in basso a sinistra (*ab*) e del punto in basso a destra (*bb*) vengono interpolate in *x2*.

```
x1 = lerp(u, grad(aa, x, y), grad(ba, x - 1, y))
x2 = lerp(u, grad(ab, x, y - 1), grad(bb, x - 1, y - 1))

return lerp(v, x1, x2)
```

x1 è l'interpolazione orizzontale tra i contributi dei due gradienti sul lato superiore della cella, ovvero quanto quei gradienti influenzano il punto *(x, y)* in base alla loro direzione e alla loro posizione relativa all'interno della cella. *x2* invece è l'interpolazione orizzontale tra i contributi dei due gradienti sul lato inferiore. Infine, *x1* e *x2* vengono interpolati tra di loro (facendo quindi una media pesata anche verticalmente) per ottenere il risultato finale, che è il valore di Perlin Noise per il punto *(x, y)*.

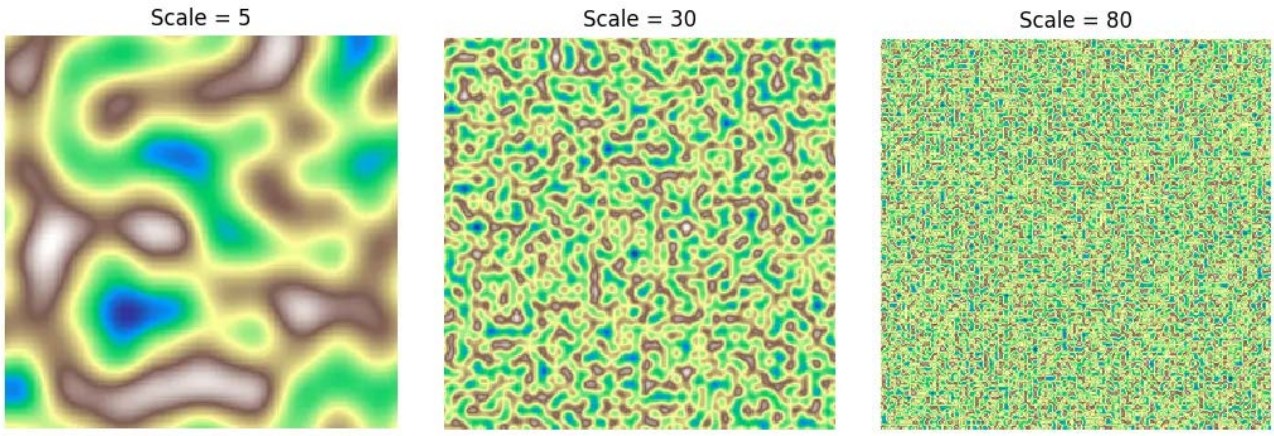
Per calcolare l'intera heightmap di valori di Perlin Noise, bisogna quindi calcolare ogni punto singolarmente:

```
def generate_perlin_noise(width=256, height=256, scale=50.0):
    """Generate a 2D array of Perlin noise."""
    noise_map = np.zeros((height, width))
    for y in range(height):
        for x in range(width):
            # Normalize the coordinates between 0 and 1,
            # indipendent from the size of the image
            nx = x / width
            ny = y / height
            value = perlin(nx * scale, ny * scale)
            noise_map[y][x] = (value + 1) / 2  # Normalize from [-1,1] to [0,1]
    return noise_map
```

La funzione prende in input una grandezza custom della heightmap da creare e poi normalizza ciascuna coordinata su cui bisogna calcolare il valore di Perlin Noise, di modo che questo metodo funzioni con qualsiasi dimensione di heightmap.

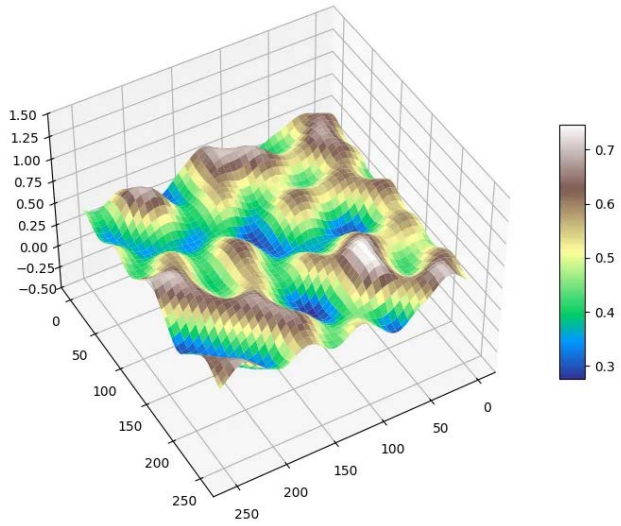
I valori x e y sono poi moltiplicati per un fattore di scala. Il parametro *scale* controlla la frequenza del rumore, cioè quanto velocemente cambia nello spazio. Un valore piccolo (e.g. 5) porterà ad un risultato con meno dettagli, con rumore che cambia in modo più graduale, indicato per esempio per la

generazione di terreno morbido o nuvole uniformi. Un valore alto invece (e.g 80) porterà ad una heightmap con tanti dettagli fini, più utile per esempio per la roccia.



Le implementazioni di Perlin noise sono diverse, in base al risultato che si vuole ottenere. I gradienti predefiniti possono essere modificati sia per ottimizzare la computazione, che per migliorare i risultati, minimizzando alcuni pattern o artefatti visibili.

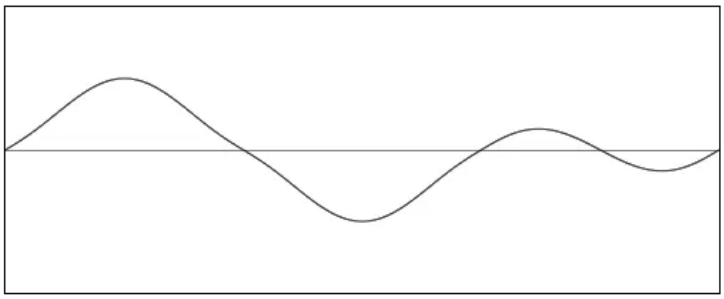
Il problema principale del Perlin Noise, comune per tutte le implementazioni di questo tipo, è che i risultati sembrano “piatti”, manca dettaglio, i gradienti sono troppo morbidi, non naturali.



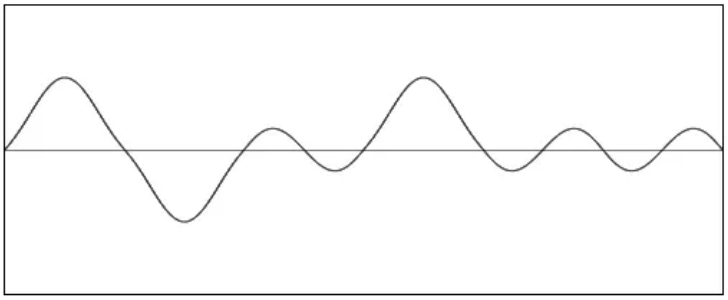
(plot 3D con scale=5)

Fractal Perlin Noise

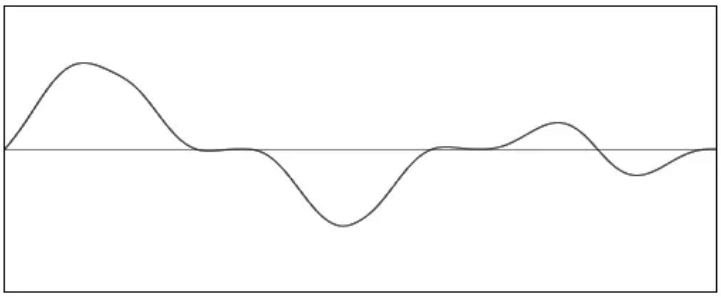
Una Perlin noise monodimensionale con frequenza 1 potrebbe avere questa forma:



mentre con frequenza 2 questa forma:

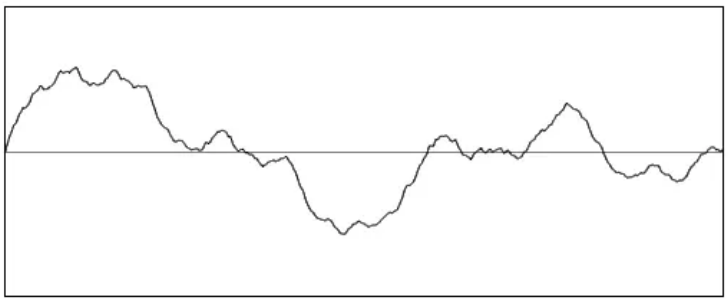


Il semplice aumento della frequenza non aggiunge dettagli reali, o naturali. La curva rimane troppo graduale e cambia valori ad una velocità fissa. Per aggiungere dei dettagli più organici, possiamo sommare le due curve, pesando le seconda 0.5:



questa si chiama Perlin noise a 2 ottave. L'utilizzo di "ottava" arriva dalla musica, dove per salire di ottava bisogna raddoppiare la frequenza. Allo stesso modo, nel Fractal Perlin Nose, spesso la frequenza viene raddoppiata.

Aggiungendo più ottave, si possono raggiungere dettagli con sempre più dettagli piccoli. Con 4 ottave:



Il Fractal Perlin Noise usa lo stesso meccanismo per la generazione della heightmap di Perlin noise, ma al posto di generare una sola mappa ne genera multiple, a scale crescenti, e le sovrappone.

Il nome "Fractal" è usato per descrivere oggetti geometrici autosimili, ovvero che hanno la stessa struttura ad ogni scala di osservazione. In questo caso il risultato avrà una auto-similarità statistica, ovvero la struttura dei valori appare simile ad ogni scala, pur non essendo identica. Questo è dato dal fatto che per ogni ottava, ovvero ad ogni scala, la heightmap è generata seguendo le stesse regole.

Dettagli Implementativi

Oltre al numero di ottave (il numero di livelli di dettaglio da sovrapporre), la funzione richiede un valore di *persistence*, ovvero quanto è importante ciascun layer successivo, rispetto al precedente. Per esempio, con un valore di 0.5, il primo layer avrà *amplitude* 1, il secondo 0.5, il terzo 0.25... così che ogni layer sempre più dettagliato avrà meno importanza nella somma finale.

```python
def fractal_noise_2d(x, y, octaves=6, persistence=0.5, lacunarity=2.0):
    total = 0
    amplitude = 1.0
    frequency = 1.0
    max_value = 0  # Used to normalize result

    for _ in range(octaves):
        value = perlin(x * frequency, y * frequency)
        total += value * amplitude

        frequency *= lacunarity
        amplitude *= persistence
        max_value += amplitude

    # Normalize the result to [0, 1]
    return total / max_value if max_value != 0 else 0
```

Quindi per ogni ottava viene aumentata la frequency in base al valore di *lacunarity*, e viene diminuita l'*amplitude* in base al valore di *persistence*, di modo che ciascun layer successivo avrà più dettagli ma sarà meno importante nella somma finale.

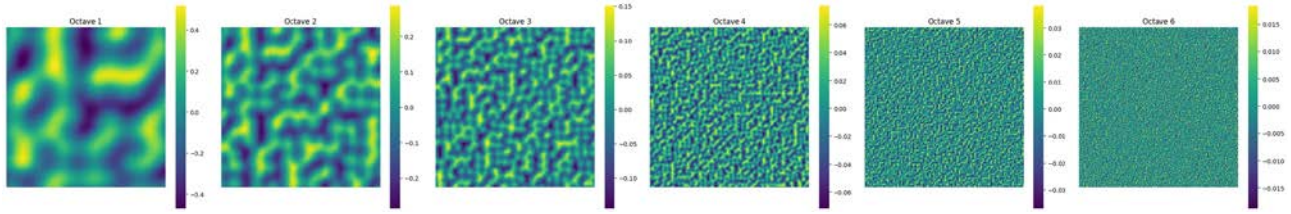
La somma tra i vari layer produce una heightmap finale fuori scala, con valori superiori a 1. Per questo motivo, viene normalizzata per riportare i valori nell'intervallo [0, 1].

Esattamente come nel Perlin Noise normale, questo procedimento viene eseguito separatamente per ciascuna coordinata (x, y) della heightmap:

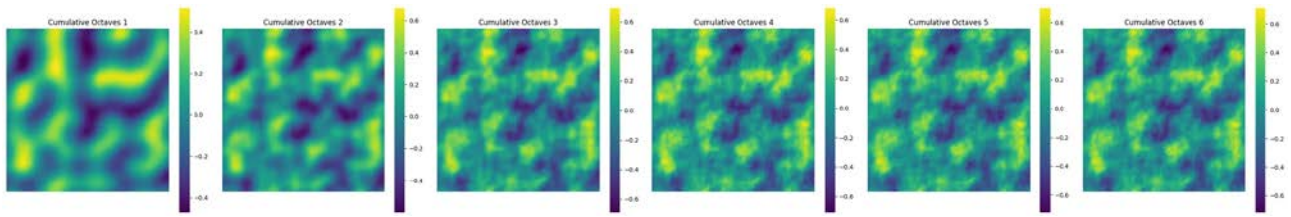
```python
def generate_fractal_noise(width=256, height=256, scale=50.0, octaves=6, persistence=0.5, lacunarity=2.0):
    noise_map = np.zeros((height, width))
    for y in range(height):
        for x in range(width):
            nx = x / width
            ny = y / height
            value = fractal_noise_2d(nx * scale, ny * scale, octaves, persistence, lacunarity)
            noise_map[y][x] = value  # Already normalized to [0,1]
    return noise_map
```

Anche qui la funzione richiede, oltre ai parametri appena descritti, un valore di scale che definisce il livello di dettaglio di partenza.

Per esempio, utilizzando 6 ottave, una scala iniziale di 5, *persistence* di 0.5 e *lacunarity* di 2, le ottave generate sono le seguenti:

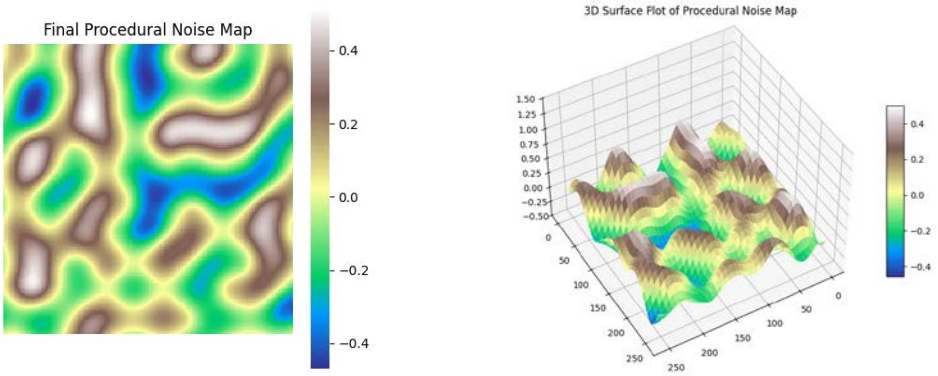


ognuna ha un livello di dettaglio crescente, raddoppiato rispetto a quello precedente. Queste vengono man mano sommate con un peso che viene dimezzato per ogni successiva ottava:

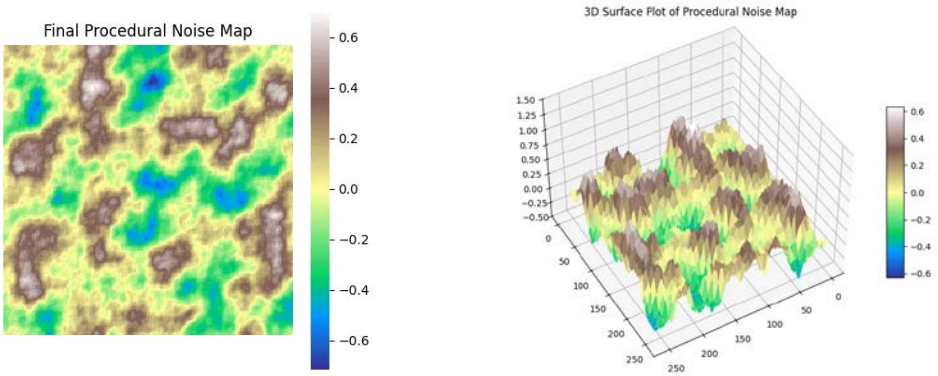


dopo la somma di ogni ottava, la quantità di dettaglio aumenta sempre di più.

Si parte da una heightmap iniziale, ottenuta applicando il Perlin Noise a una singola ottava:



Sommando sei ottave, si ottiene il seguente risultato, decisamente più ricco di dettagli:

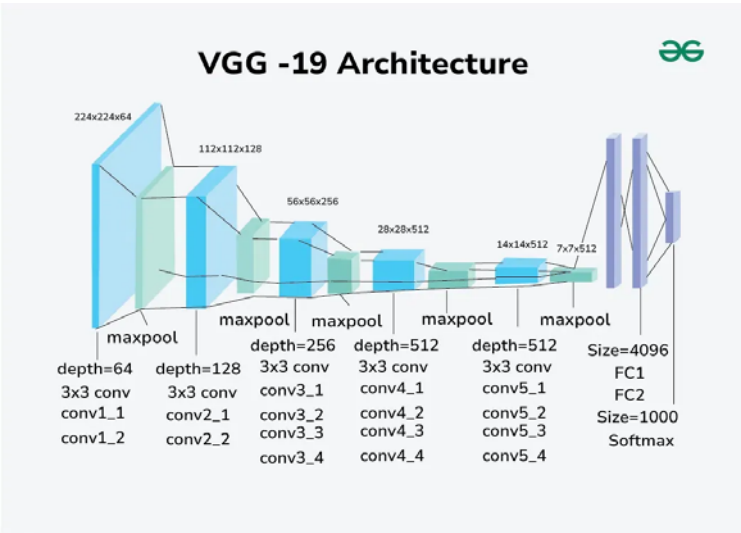


Il risultato è molto più realistico, molto più dettagliato e meno irrealisticamente morbido. Variando i parametri è possibile adattare il risultato alle proprie esigenze, in base a ciò che si vuole generare.

Style Transfer

La tecnica di Naural Style Transfer (NST) può essere applicata ai risultati dell'Explicit Noise o del Perlin Noise per applicare lo stile di un terreno reale. Diversi tipi di heightmaps reali possono essere usate per avere risultati diversi in base all'use case, per esempio la heightmap di un fiume, oppure quella di un particolare tipo di montagna.

Viene usata la rete neurale convoluzionale VGG-19, pre-trainata su ImageNet, come feature extractor:



La VGG-19 ha 19 strati, di cui 3 finali fully connected. Il layer di classificazione finale non è necessario in questo caso e viene quindi tagliato. I layer convoluzionali sono divisi in 5 gruppi diversi, separati da operazioni di maxpooling, utili a ridurre il numero totale di parametri. Le attivazioni di alcuni layer convoluzionali saranno utilizzate per i calcoli della loss function per applicare lo style transfer.

Questa tecnica non necessita di training o di fine tuning, viene fatto tutto in fase di inferenza, andando ad ottimizzare l'immagine in modo diretto, ovvero modificandone i pixels iterativamente.

L'obiettivo dell'ottimizzazione è quello di ridurre la loss function che misura quando l'immagine di output sia vicina al contenuto dell'immagine di partenza (*content image*) e allo stile dell'immagine di riferimento (*style image*). La loss function è composta da 3 diverse loss, pesate.

Content Loss

Il processo di ottimizzazione deve preservare la forma e la struttura generale della heightmap generata proceduralmente. La content loss misura la similarità tra l'immagine di output (ovvero quella modificata dopo l'ottimizzazione) e la *content image*, ovvero l'immagine generata tramite Perlin noise.

$$L_{content} = \sum (F_{output} - F_{content})^2$$

L'operazione viene fatta usando le attivazioni di *conv5_2*, ovvero il secondo layer convoluzionale del quinto gruppo, perchè questo livello profondo cattura features semantiche ad alto livello, non solo bordi o colori ma invece forme e strutture generali.

Di conseguenza per fare questo calcolo, la *content image* è fatta passare per la rete una volta, salvando le attivazioni di questo layer. Queste saranno confrontate tramite la loss function con le attivazioni dello stesso layer dopo aver fatto passare l'immagine di output dell'ultima ottimizzazione.

Style Loss

Lo scopo della Style Loss è quello di trasferire le caratteristiche della *style image* nella output image, in termini di texture e patterns. In questo caso i layer utilizzati sono conv1_1, conv2_1, conv3_1, conv4_1 e conv5_1, ovvero il primo layer convoluzionale di ciascun blocco.

Le correlazioni tra le attivazioni dei diversi layers di ciascuna immagine sono calcolate tramite la matrice di Gram. Ogni elemento della matrice rappresenta quanto sono simili (cioè quanto si correlano) i filtri *i* e *j* su tutta l'immagine. Questo permette di rappresentare lo stile dell'immagine, indipendentemente dalla posizione, catturando pattern simili a textures e relazioni statistiche.

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l$$

dove *F* è l'attivazione del filtro *i* nella posizione *k* del layer convoluzionale *l*.

Sono quindi calcolate 2 matrici di Gram: una per l'output image e una per la style image.

$$L_{style} = 1/4 N_l^2 M_l^2 \sum_{ij} (G_{output} - G_{style})^2$$

dove *N* è il numero di filtri e *M* è il numero di posizioni spaziali del layer *l*.

La differenza tra le 2 matrici è quindi calcolata tramite Mean Squared Error (MSE) e normalizzata da un fattore che prende in considerazione sia il numero di filtri che le dimensioni spaziali.

La comparazione effettuata a diversi livelli della rete è utile a confrontare features di basso e alto livello, e questa normalizzazione permette di considerare i diversi layers nello stesso modo, a prescindere dal numero di filtro o le dimensioni spaziali.

Total Variation (TV) Loss

L'obiettivo della Total Variation Loss è quello di ridurre gli artefatti causati dall'ottimizzazione e rendere l'immagine di output più liscia, incoraggiando la coerenza tra valori di pixel vicini.

$$L_{TV} = \sum_{i,j}(||x_{i,j} - x_{i+1,j}|| + ||x_{i,j} - x_{i,j+1}||)$$

quindi per ogni pixel dell'immagine di output, calcola la differenza assoluta tra il pixel e quello sotto, e tra il pixel e quello alla sua destra.

Più i pixel sono simili e più bassa sarà questa loss, quindi più liscia apparirà l'immagine.

Final Optimization Loss

Le 3 loss functions vengono concatenate, con pesi diversi, in una loss function unica.

$$Loss = \alpha L_{content} + \beta L_{style} + \gamma L_{TV}$$

Questi parametri sono stati impostati come segue:

α = 1*10^−5, β = 2.5*10^−11, γ = 1*10^−10

Processo di Ottimizzazione

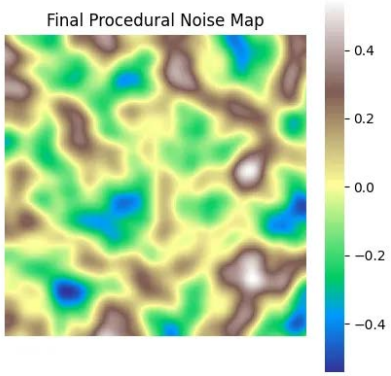
Ricapitolando, il processo di ottimizzazione completo è:

- genera una heightmap tramite Perlin Noise o Explicit Noise (*Content Image*),
- ottieni una heightmap reale con lo stile desiderato (montagne, fiumi...) (*Style Image*),
- estrai le attivazioni dei layers necessari della rete neurale convoluzionale, rispetto alle due heightmap,
- esegui il ciclo di ottimizzazione partendo dalla content image e aggiornando l'immagine utilizzando la discesa del gradiente stocastica (SGD) e la Final Optimization Loss, con un learning rate decrescente.

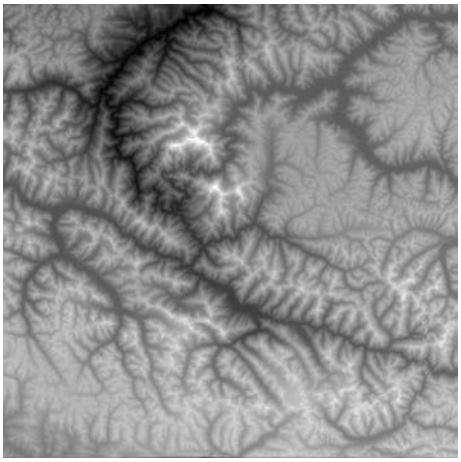
Questo processo è relativamente veloce, è eseguibile tramite Colab T4 GPU impiegando 2 minuti per 500 iterazioni di ottimizzazione, che sono sufficienti per raggiungere un ottimo risultato.

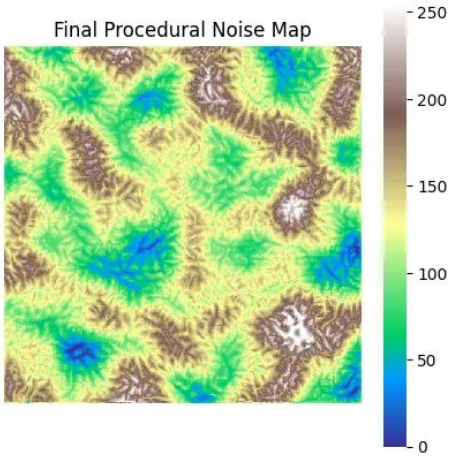
Risultati

Una heightmap generata tramite Perlin Noise con 2 ottave ha questo aspetto:



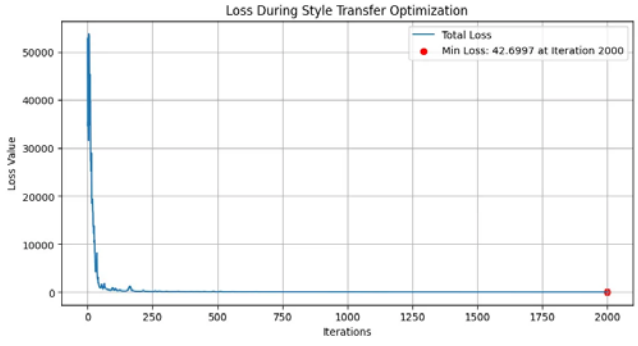
Applicando la seguente heightmap stilistica rappresentante le montagne dell'Himalaya, si raggiunge questo risultato dopo 500 iterazioni:

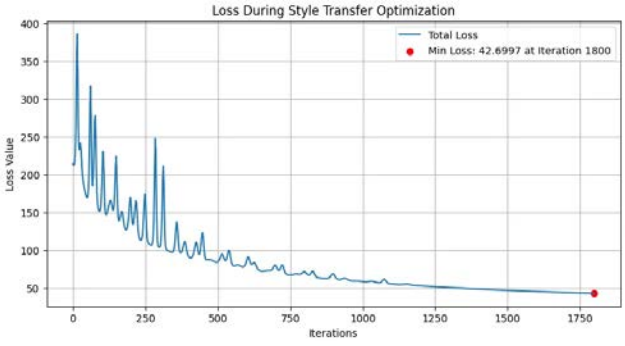




Il risultato è molto più dettagliato e realistico. La heightmap contiene ora lo stile dell'immagine di riferimento.

La loss decresce in questo modo (a sinistra quella completa, a destra tagliando i primi 200 valori):





Una cosa importante da notare è che questo approccio causa la perdita della capacità di generare la mappa in tempo reale, chunk per chunk. Questo è dato dal fatto che la loss function viene calcolata sull'intera immagine, e di conseguenza sarebbe diversa se l'immagine avesse dimensioni differenti, e gli step di ottimizzazione sarebbero anche loro diversi.

StyleTerrain: Modello Generativo Disentangled per Terreno Procedurale

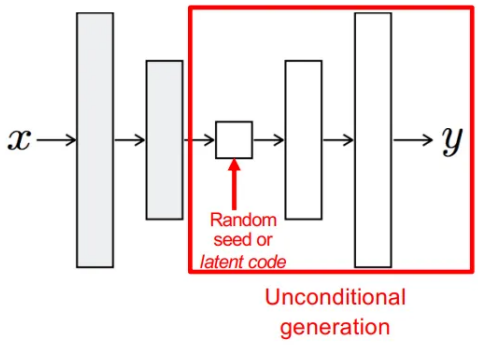
Nella generazione procedurale di terreni è importante bilanciare tra il realismo, catturando le features dello stile che si vuole riprodurre, e il costo di creazione. La mancanza di controllabilità dei risultati dei metodi basati sul machine learning presenta una delle maggiori sfide attuali per questo tipo di metodi.

Introduzione alle GAN

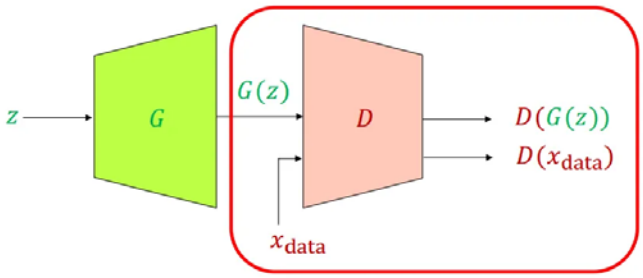
Sviluppi recenti della generazione di terreni utilizzano metodi basati sul machine learning, specialmente sulle *Generative Adversarial Networks (GANs)*.

Le GAN sono un tipo di rete neurale usata per la generazione di immagini, imparando a campionare dalla distribuzione del training set. Possono anche essere usate delle labels per dividere il training set in categorie e poi controllare l'output. In questo caso di parla di *conditional generation*. Possono anche essere usate per trasformare un'immagine di input secondo un certo stile.

L'idea del meccanismo di generazione arriva dagli autoencoders. Nel caso di generazione non condizionale, viene utilizzata solo la parte a destra, partendo da un latent code random oppure campionandolo da una distribuzione. Invece, se vogliamo applicare un cambiamento ad un'immagine, viene utilizzata l'intera rete.



La rete di una GAN è composta da un generatore e un discriminatore.



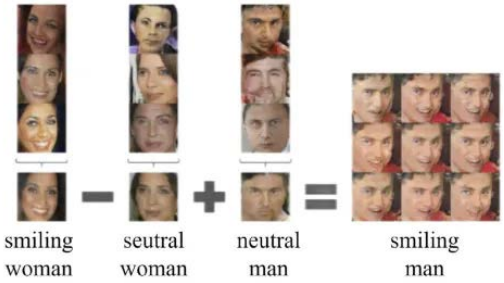
Per il training della rete viene utilizzato il meccanismo "adversarial", dove il generatore e il discriminatore competono. Il generatore è la parte della rete che vuole imparare a generare i samples, mentre il discriminatore vuole imparare a distinguere tra i samples reali e quelli generati.

Durante il training, i gradienti passano dal discriminatore al generatore, aggiornandone i parametri.

In questo processo, il generatore si comporta come una black box dal punto di vista del discriminatore: quest'ultimo valuta le immagini generate senza conoscere come vengano prodotte. Il generatore cerca di minimizzare la loss ingannando il discriminatore, mentre quest'ultimo cerca di massimizzarla distinguendo correttamente le immagini reali da quelle sintetiche.

Il discriminatore può essere rimosso dopo il training, mantenendo solo il generatore a cui viene dato in input un latent code z che può essere random o campionato da una distribuzione.

Il latent code può essere manipolato, per esempio applicando operazioni aritmetiche:



oppure interpolazioni:

GAN Disentanglement: Variante di StyleGAN2

Nel caso di generazione condizionale (dove un label è associato a ciascun sample), è necessario avere un dataset ben definito, il che richiede molto tempo e complessità specialmente nel caso ciascun sample possa avere più di una caratteristica. Di conseguenza, la performance dipende pesantemente dalla qualità del feature engineering applicato al dataset.

Per risolvere questo problema si usa il representation learning, dove deve essere il modello stesso a estrarre le features dai dati in modo autonomo.

Il representation learning comprende un metodo chiamato "*disentangled representation learning*", che permette il controllo dell'output generato attraverso i codici latenti.

L'idea è quindi quella di poter manipolare il codice latente rispetto a determinati fattori, lasciando gli altri invariati. Il modello deve essere in grado di separare i fattori di variazione nascosti (latent factors) che influenzano l'output generato, in modo che ogni dimensione nello spazio latente rappresenti un aspetto semantico distinto.

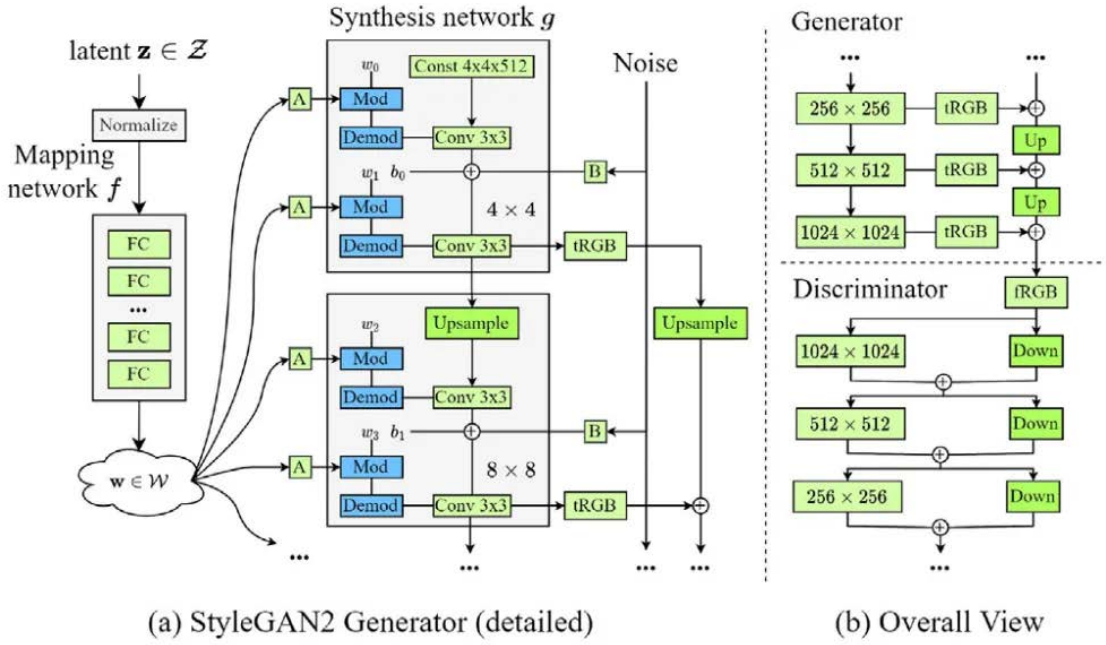
Ad esempio, in un GAN per la generazione di volti:

- una dimensione potrebbe controllare il sesso,
- un'altra il colore degli occhi,

- un'altra ancora la presenza di barba o capelli lunghi.

Nelle GAN tradizionali lo spazio latente *z* è spesso altamente *entangled* (aggrovigliato), di conseguenza piccole variazioni possono causare cambiamenti globali non interpretabili nell'immagine. Il *disentanglement* cerca di risolvere questo problema attraverso diverse strategie.

Il generatore di terreni è chiamato *StyleTerrain*, dato che adatta la rete StyleGAN2 per la generazione di heightmap di terreni a 16-bit. StyleGAN2 introduce una doppia rete composta da un *mapping network* e un *synthesis network*.



Mapping Network

Il mapping network è una rete feed-forward che trasforma il vettore latente iniziale *z* in uno spazio intermedio *W*. Questo processo permette di disaccoppiare i fattori di variazione presenti in *z*, ottenendo un codice *w* che è più interpretabile e controllabile.

Synthesis Network

Il synthesis network genera l'immagine a partire da *w*, usando operazioni di style transfer a diversi livelli di risoluzione, aumentando la risoluzione ad ogni livello. Quindi, con l'aiuto del mapping network, la rete può imparare un mapping che divide l'entanglement delle features.

Ogni blocco del synthesis network corrisponde ad un livello di risoluzione specifico (4x4, 8x8...). Il processo inizia con una feature map di dimensione piccola (4x4x512) che è costante e imparata durante il training. Questo è il punto di partenza per costruire la heightmap progressivamente.

Ad ogni livello, il synteshis network applica convoluzioni per elaborare la feature map precedente. I pesi di ciascuna convoluzione sono modulati per un fattore w_i ricavato dal codice latente *w*. Questo applica lo style transfer tramite la modulzione dei pesi della convoluzione, rispetto ai valori del codice latente *w*, permettendo di cambiare lo stile dell'immagine a quel livello.

Grazie alla disaccoppiatura dello spazio *W*, ogni w_i può influenzare aspetti specifici dell'immagine (pendenza, rugosità...). Questo consente manipolazioni precise tramite interpolazione o style mixing.

La demodulation è una forma di normalizzazione delle feature map, introdotta in StyleGAN2 per ridurre gli artefatti visivi causati dalla modulation, dato che la modulazione moltiplica i pesi per un fattore, il che può causare variazioni improvvise nella scala delle feature map che causano distorsioni o artefatti visivi.

Inoltre, viene aggiunto del rumore b_i tra le due convoluzioni di ogni blocco (campionato da una distribuzione normale), per introdurre variazioni localizzate nella feature map, ovvero dettagli fini e variazioni casuali nell'immagine generata.

Ad ogni livello, si applica un upsampling per raddoppiare la risoluzione della feature map interna. Al termine di ogni blocco, si applica una trasformazione "*to RGB*" (tRGB) per convertire la feature map in immagine, producendo un risultato parziale ad una determinata risoluzione.

L'upsampling esterno ai blocchi permette di mescolare le informazioni provenienti da diversi livelli di risoluzione. Ciò garantisce che gli attributi semantici controllati da w_i siano coerenti a tutte le scale. Ad esempio, se w_i controlla la pendenza globale, essa deve essere presente sia nelle feature map di bassa risoluzione che in quelle di alta risoluzione. L'upsampling esterno assicura che queste informazioni vengano fuse correttamente.

La heightmap finale è data dalla somma di tutte le immagini parziali generate dai diversi livelli.

Il generatore è suddiviso in livelli progressivi , dove ogni livello aggiunge dettagli a una certa scala:

- i livelli più bassi (da 0 a 3) generano la struttura grossolana (layout generale del terreno),
- i livelli intermedi (dal 4 a 7) aggiungono dettagli come colline o fiumi,
- i livelli alti (da 8 a 17) si occupano dei dettagli fini (rugosità, texture).

Lo spazio *W* è fondamentale perché permette manipolazioni lineari che producono cambiamenti localizzati e interpretabili nell'output. Per esempio:

1. Interpolazione tra due punti in *W:*

- Due codici *w*1 e *w2* rappresentano due tipi di terreno.
- Interpolando linearmente tra *w*1 e *w*2 , si ottiene una transizione fluida tra i due stili di terreno (es. colline → montagne).

2. Style mixing

- Si prende parte del codice *w*1 (ad esempio, relativo alla forma grossolana del terreno) e parte del codice *w*2 (dettagli fini come fiumi).
- Il risultato è un terreno che combina caratteristiche dei due input.

3. Editing mirato

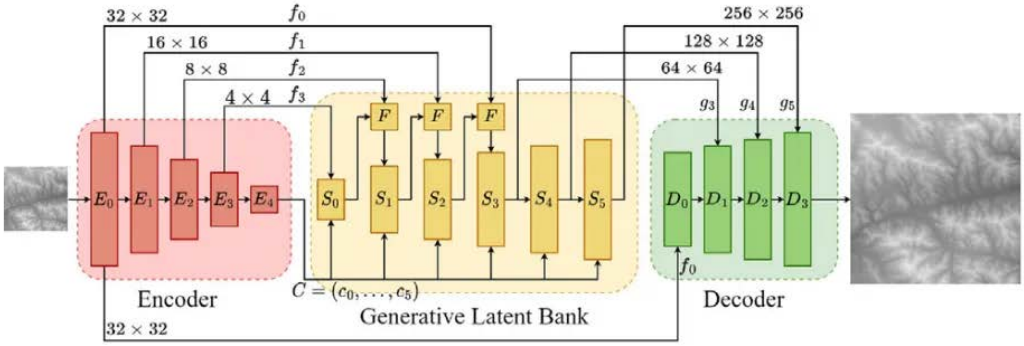
- Cambiando solo alcune dimensioni di *w*, si può modificare selettivamente un aspetto del terreno (es. aumentare la pendenza senza alterare altro).

Tutte queste operazioni possono essere effettuate nello spazio latente intermedio *W*.

GLEAN (Generative LatEnt bANk)

GLEAN è un modello avanzato di super-risoluzione delle immagini (aumentare la risoluzione delle immagini), mantenendo coerenza semantica e dettagli realistici. In StyleTerrain , GLEAN viene utilizzato per migliorare la risoluzione delle heightmap generate dal generatore StyleGAN2, permettendo di ottenere mappe di altezza fino a 8192×8192 pixel , con dettagli fini e coerenti.

GLEAN si basa sull'idea di utilizzare un generatore pre-addestrato (come StyleGAN2) come banca di pesi latenti per il processo di super-risoluzione. Invece di addestrare una rete completamente nuova per la super-risoluzione, GLEAN riutilizza i pesi del synthesis network di StyleGAN2, aggiungendo nuovi strati convoluzionali per integrare informazioni provenienti da un encoder che analizza l'immagine in input.



L'encoder prende in input l'immagine a bassa risoluzione ed estrae un codice latente *C* e una serie di feature map multirisoluzione f_i (una per ogni livello del synthesis network).

Il codice latente *C* rappresenta una descrizione compatta dell'immagine, è una sorta di descrizione astratta della struttura generale dell'immagine.

Le feature maps sono estratte a diverse risoluzioni durante l'encoding. I livelli più bassi catturano per esempio i bordi, mentre quelli più alti catturano strutture più complesse come pendenze o forme generali del terreno. Le feature maps vengono usate per fondere informazioni locali e globali con quelle generate dal synthesis network.

Il generatore centrale condivide i pesi con il synthesis network di StyleGAN2. Questo viene usato come una "banca di pesi" pre-addestrata per generare feature maps ad alta risoluzione g_i. Le feature map f_i estratte dall'encoder vengono fuse con le feature map g_i prodotte dal synthesis network ad ogni livello, in modo da mantenere sia le informazioni semantiche dell'input originale sia i dettagli realistici forniti dal synthesis network.

Infine, il decoder costruisce l'immagine finale ad alta risoluzione, combinando le informazioni estratte dall'input a bassa risoluzione con quelle generate dal synthesis network tramite convoluzioni, per raffinare le informazioni, ed aumentando progressivamente la dimensione spaziale dell'immagine.

Metriche di Performance e Risultati

Nel paper vengono utilizzate due principali metriche per valutare le performance del modello proposto.

Quantitative metric scores.

Dataset	FID	Full path length
FFHQ 1024 × 1024 RGB	5.61	151.44
FFHQ 1024 × 1024 GRAY	4.24	225.80
ASTERGDEMV3 (ours)	5.67	**47.42**

Perceptual Path Length (PPL)

Il PPL è una metrica specificamente utilizzata per valutare il grado di disentanglement dello spazio latente di un GAN, dato che uno spazio latente molto separabile permette un controllo fine sui fattori della generazione:

- dati due punti nello spazio latente intermedio w1 e w2, viene eseguita un'interpolazione lineare tra loro, producendo una serie di punti intermedi,
- ogni punto intermedio viene passato al generatore per produrre un'immagine,
- viene calcolata la distanza percettiva (basata su embedding VGG16) tra immagini consecutive nell'interpolazione.

Il PPL è la somma di queste distanze. Un PPL più basso indica uno spazio latente meglio disentangled, dove piccole variazioni in una dimensione corrispondono a cambiamenti controllati e localizzati nelle immagini generate, non ci sono salti improvvisi.

Fréchet Inception Distance (FID)

La FID è una metrica utilizzata per valutare la qualità delle immagini generate da modelli di generazione come i GAN. Misura la somiglianza tra due set di immagini (quelle reali e quelle generate) calcolando la *distanza di Fréchet* tra le distribuzioni delle feature estratte dalle immagini:

- le immagini reali e quelle generate vengono passate a un modello pre-addestrato, estraendo features di alto livello
- la FID calcola la distanza di Fréchet tra le due distribuzioni approssimate tramite le feature estratte

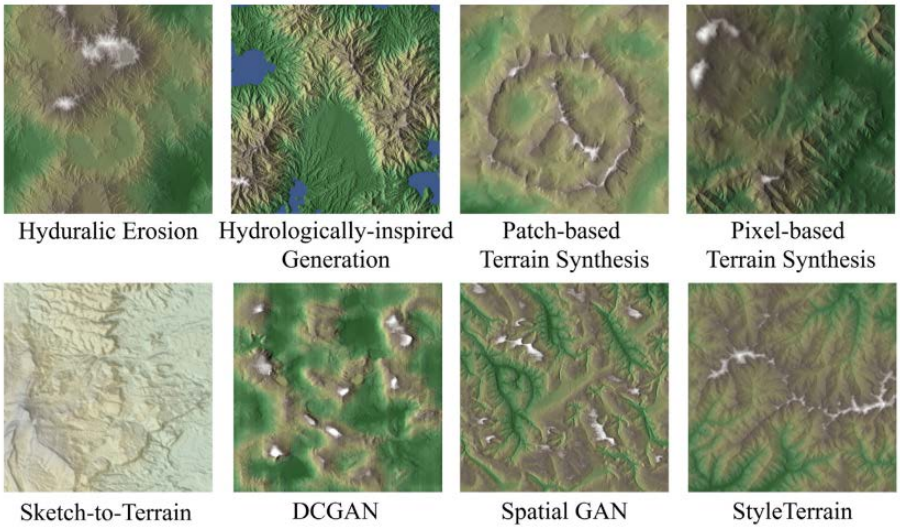
Una FID più bassa indica che le immagini sintetiche sono più simili alle immagini reali.

Risultati

Il paper non fornisce confrontati con altri metodi basati su GAN per la generazione di terreni. Questo probabilmente è dovuto al fatto che:

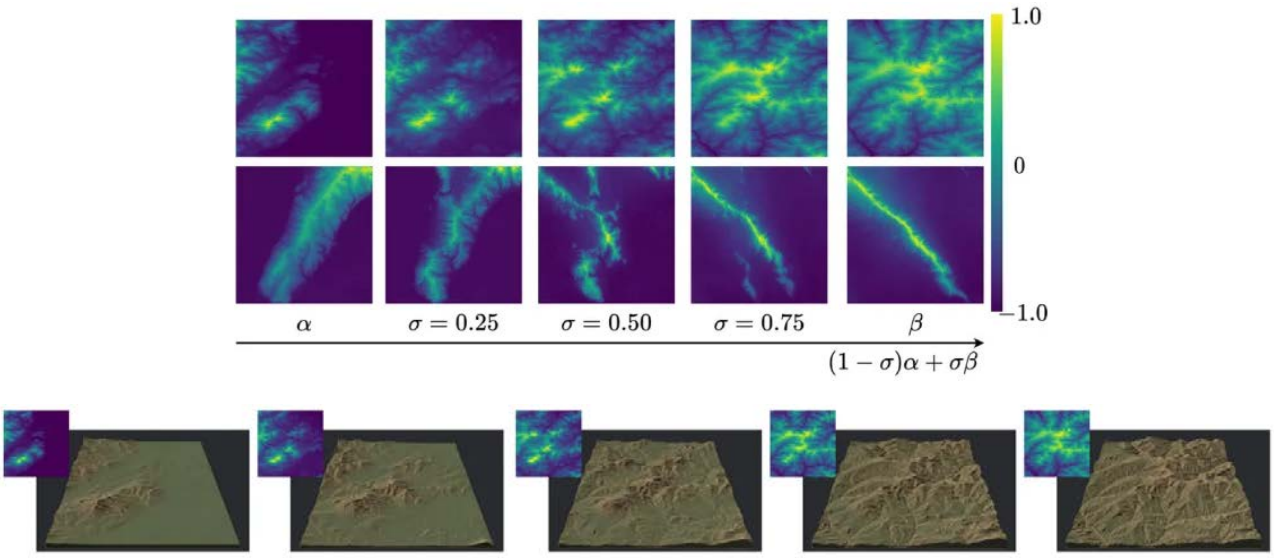
- Questa è stata la prima applicazione del disentanglement applicato ai terreni. Il modello è quindi specializzato per questo use case, e non avrebbe senso compararlo ad altre GAN disentangled per esempio che generano volti.
- Un punto di forza di questo metodo è il livello a cui si possono manipolare i risultati, cosa che non era possibile fare precedentemente.
- L'applicazione di GAN alla generazione di heightmap è ancora un campo emergente, e pochi lavori forniscono codice o modelli riproducibili per un confronto diretto.
- Molti metodi precedenti non riescono a generare immagini stabili ad alta risoluzione.

La comparazione è stata effettuata a livello qualitativo:



Il paper non entra in comparazioni dettagliate, si limita ad introdurre ciascun metodo. Viene solamente detto come altri metodi basati su GAN hanno molta instabilità nel training quando vengono utilizzati dataset con immagini ad alta risoluzione.

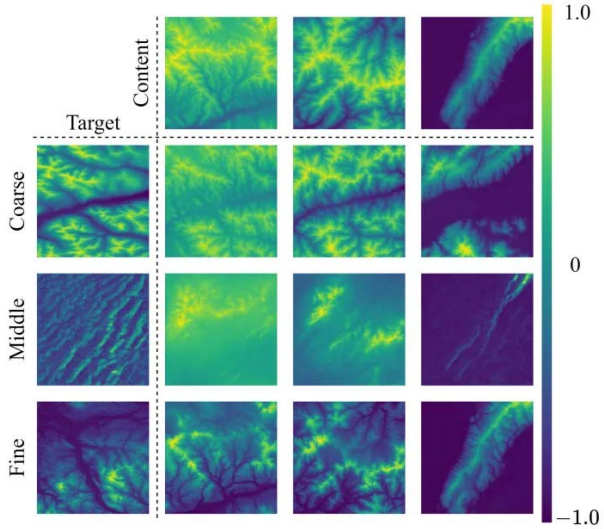
Le analisi più interessanti sono quelle relative al livello di controllabilità dei risultati. Attraverso il disentanglement dello spazio latente, è possibile interpolare tra due terreni per creare transizioni naturali:



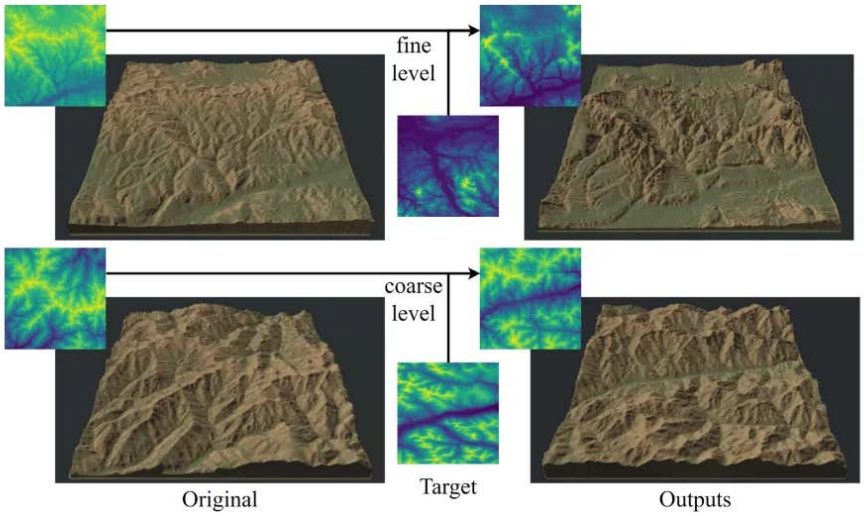
È Anche possibile mixare stili a diverse scale (coarse/middle/fine) per costruire nuovi terreni combinando caratteristiche di quelli esistenti.

Per ciascuna coppia di immagini content e target, vengono calcolati i latent codes intermedi *w1* e *w2* e vengono applicati a diversi livelli del generatore per creare un nuovo terreno ibrido. Ad esempio:

- Per la riga Coarse, si usa *w*2 (Target) nei livelli coarse (0-3), mentre *w*1 (Content) controlla tutti gli altri livelli.
- Per la riga Middle, si usa *w*2 (Target) nei livelli middle (4-7), mentre *w*1 (Content) controlla tutti gli altri livelli.
- Per la riga Fine, si usa *w*2 (Target) nei livelli fine (8-17), mentre *w*1 (Content) controlla tutti gli altri livelli.



Possiamo infatti vedere per esempio nella prima colonna della griglia interna, la prima immagine di output mantiene i dettagli con lo stile dell'immagine originale (content), applicati però alla struttura generale dell'immagine target, perchè viene usato w2 nei primi livelli del generatore (coarse). L'opposto invece può essere notato nell'ultima immagine della stessa colonna, qui la struttura generale è quella dell'immagine originale mentre i dettagli sono nello stile dell'immagine target.



Conclusioni

L'analisi dei due paper ha mostrato approcci innovativi per la generazione procedurale di terreni, con particolare attenzione a realismo, qualità e controllo sui risultati. Il primo lavoro, *Procedural terrain generation with style transfer (2024)*, combina tecniche procedurali tradizionali come il Perlin Noise con il Neural Style Transfer, permettendo di ottenere heightmap stilizzate e personalizzabili con un basso costo computazionale, utili soprattutto in ambiti creativi come il game design. Il secondo, *StyleTerrain (2023)*, introduce un modello basato su una variante di StyleGAN2 che genera terreni ad alta risoluzione con un'elevata controllabilità grazie al disentanglement dello spazio latente. Questo consente modifiche mirate e interpretabili, adatte a contesti dove è richiesta precisione nella progettazione del paesaggio. Entrambi i metodi rappresentano un passo avanti nella creazione di ambienti virtuali più realistici e flessibili, fondendo le potenzialità delle tecniche procedurali con quelle di machine learning.

Sorgenti

- *Procedural terrain generation with style transfer*: https://arxiv.org/abs/2403.08782
- *Perlin Noise: A Procedural Generation Algorithm*: https://rtouti.github.io/graphics/perlin-noise-algorithm
- *How Does Perlin Noise Work?*: https://youtu.be/9B89kwHvTN4?si=AWqEcFdwntFpwk3l
- *Improving Noise*: https://mrl.cs.nyu.edu/~perlin/paper445.pdf
- *How to turn a few Numbers into Worlds (Fractal Perlin Noise)*: https://youtu.be/ZsEnnB2wrbI?si=a2CAdGYKncVmHjF4
- *StyleTerrain: A novel disentangled generative model for controllable high-quality procedural terrain generation*: https://www.sciencedirect.com/science/article/pii/S009784932300225X