Генерація Ландшафту. Звіт

Artem Onyschuk, Arseniy Stratyuk, Taras Levytskiy, Taras Kopach

8 травня 2025 р.

UI

Для створення інфтерфейсу в основному була використана бібліотека PyQt5, а також OpenGL. Він максимально простий і складається з двох станів: Головного меню і Головного інтерфейсу. Стани переключаються за допомогою кнопки на голоному меню. Стан Головного інтерфейсу ділиться знову ж таки на дві частини: бічне меню та вікно візуалізації ландшафту.

Меню

На даному стані зображений логотип проєкту та кнопка Start, яка ініціалізує генерацію при seed = 1

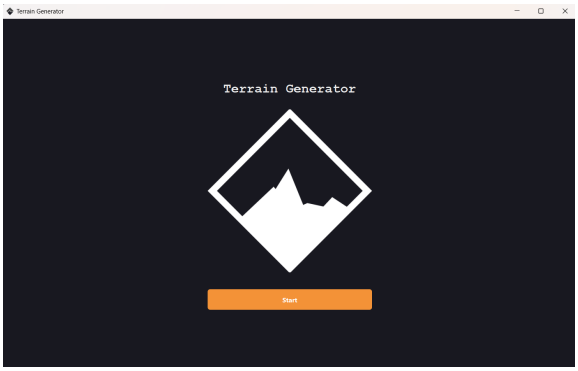


Рис. 1: Екран Головного меню

Бічне Меню

Коли екран переходить у стан головного інтерфейсу, то користувач може побачити зліва меню конфігурації створення ландшафту. Для змінення досутпні такі параметри:

- Seed - Встановлює псевдорандом для нашого генератору, приймає лише цілі числа.
- Object Intensity - Частота появи об'єктів, від 0.0 до 1.0.
- Rings - Розмір карти $3^{n_{rings}}$, концептуально формує кількість кілець від центру, не може бути менше ніж 1.
- Generation Rate - частота скільки тіків потрібно для створення нового чанку, тіків 15 на секунду, не може бути менше ніж 1.
- Height Intensity - Інтенсивність висоти.

Слайдери параметрів - власно створені класи згрупованих компонент бібліотеки PyQt. За потребою бічне меню можна приховати посунувши його вліво. При натисканні кнопки Generate приймаються параметри й сід та запускається заново генерація. У разі відсутності сіда використоується стандартний сід = 1.

Вікно Генерацій

Основне вікно, основою якого є бібліотека OpenGl. При натисканні кнопки Generate ререндириться вікно і запускається initializeGL(). Таким чином формуються регіони, об'єкти та ландшафт.

Генерація Регіонів

Для генерації регіонів був використаний простий алгоритм. Його можна зообразити автоматом

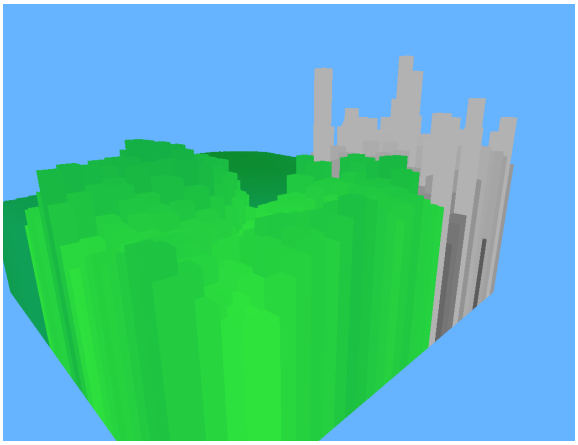
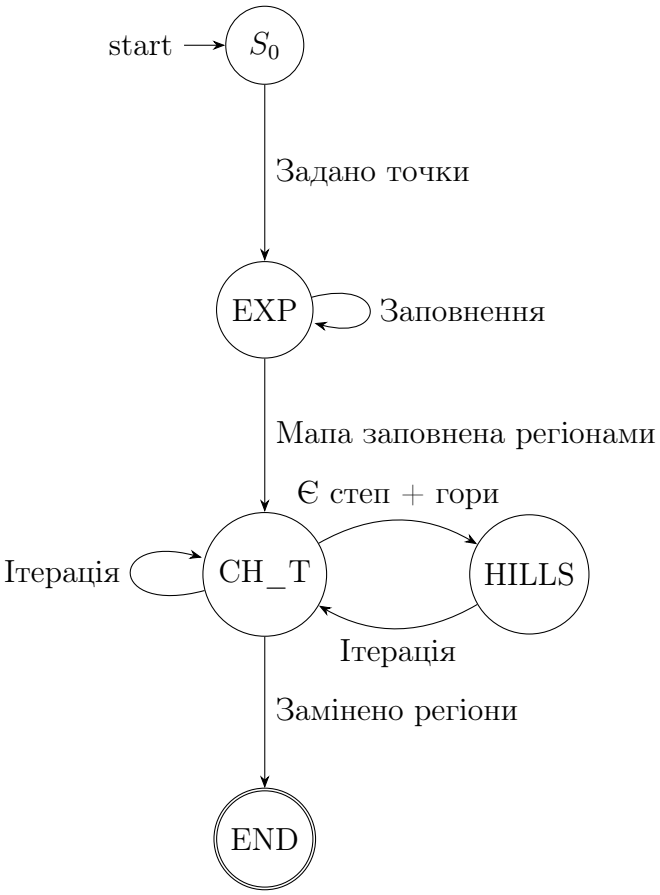


Рис. 2: Екран Візуалізації



Генерація регіонів починається з ініціалізації словника, де ключі - координати, а значення - регіон. Загалом серед регіонів є степ, сніжний степ, горбата рівнина та гірські висоти. Також фіксується розмір світу змінною border. Далі випадковим чином на карті вибираються декілька точок які будуть мати певний регіон, створюється черга. Метод перезодить у стан EXP (expand) Випадковим чином вибирається блок та його сусід, який буде перемайти регіон.
Як тільки черга буде пустою метод перезодить у стан CH_T (Check Transitions). Головна ідея в тому, щоб між регіонами гір та степу був перехід у вигляді горбартої рівнини. Ітеруючись по всій карті, ми дивимось на сусдів блоків, і якщо вони мають сусідів СТЕП і ГОРИ, тоді ми задаєм цей блок на ГОРБИ включно з його сусідами. Пісня цього циклу метод переходить у стан END. Повертається той самий словник з даними, який далі використовуються для генерації

Візуалізація

FSMT-gen візуалізує генерацію за допомогою бібліотеки OpenGL. Спершу програма генерує дані залежно від заданих параметрів в такому порядку: спершу генеруються регіони, тоді висота кожного блоку, і тоді об'єкти. Тоді вже відбувається сама генерація - поступово з'являються чанки, швидкість створення яких визначається параметром generation_rate. Як тільки кількість чанків в світі дорівнює $3^{n-rings}$, нові чанки перестають створюватись.

Чанки

Світ ділиться на чанки, кожен з яких має по 9 блоків (кожен блок має радіус 1). Власне в чанках зберігаються буфери (про це в наступній секції) для кожного блоку й об'єкту всередині чанку. Якщо чанк знаходиться в процесі створення або всередині нього вибрано блок, відповідні буфери оновлюються з високою частотою. В інших випадках буфери залишаються неактивними і не оновлюються, що дозволяє зменшити навантаження на GPU.

VAO/VBO/EBO

Дані для 3Д візуалізації зберігаються в трьох буферах - VAO (vertex array object, зберігає конфігурацію прив'язок буферів і атрибутів), VBO (vertex buffer object, зберігає інформацію про вершини й їхні дані), EBO (element buffer object, зберігає індекси трикутників). За допомогою GLSL шейдерів трикутники "заповнюються"залежно від даних, які зберігаються в VAO. Шейдерам точки (vert)

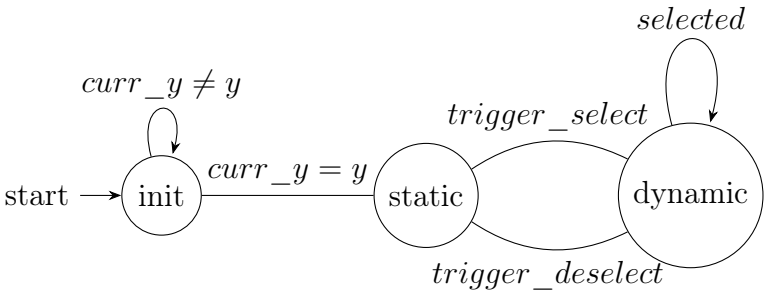


Рис. 3: Статус чанків

передаються наступні дані - позиція точки, час створення блоку, регіон блоку, чи вибраний блок, матриця моделі, теперішній час, а також матриці огляду й проєкції.

Камера

Камера в генераторі зберігає дані куту й позиції юзера, й дає шейдерам дві матриці - огляду (задає напрямок і місце спостереження) й проєкції (формує перспективу). Шейдери накладають ці матриці аби було помітно зміни.
Для повороту камерою потрібно затиснути ПКМ й повернути мишкою. Для управління використовується WASD і Shift (вгору) з Ctrl (вниз). Аби зменшити FOV й приблизити огляд можна натиснути C. Також можна вибрати блок аби вивести інформацію про нього в консоль - для цього пускається промінь, який трансформує позицію курсора мишки, позицію й кут огляду камери й шукається перетин з усіма блоками.

Опис ідеї

В нашому проєкті однією з фундаментальних ідей є визначення висоти кожного паралелепіпеда. Це ми реалізували за допомогою **Perlin Noise** — алгоритму для створення плавного, природного псевдовипадкового шуму, який часто використовується у графіці та процедурній генерації.

Принцип роботи Perlin Noise

1. **Розбиття простору на кубики.**
   Простір, в якому працює Perlin Noise, розбивається на сітку з однакових кубів (у 3D).

2. **Призначення градієнтів.**
   У кожній вершині куба задається випадковий градієнт — одиничний вектор випадкового напрямку.
   Позначимо точку у просторі як $\mathbf{P} = (\mathrm{x}, \mathrm{y}, \mathrm{z})$.

3. **Обчислення векторів до вершини.**
   Для кожної з 8 вершин куба (в 3D) визначається вектор від цієї вершини до точки $\mathbf{P}$:
   $\mathbf{d_i} = \mathbf{P} - \mathbf{V_i}$,
   де $\mathbf{V_i}$ — координати i-тої вершини куба.

4. **Добуток скалярів.**
   Для кожної вершини куба обчислюється скалярний добуток між градієнтом у цій вершині $\mathbf{g_i}$ і вектором $\mathbf{d_i}$:
   $S_i = g_{ix}d_{ix} + g_{iy}d_{iy} + g_{iz}d_{iz}$

5. **Інтерполяція значень.**
   Значення S_i інтерполюються для отримання єдиного значення у точці $\mathbf{P}$.
   Для інтерполяції використовують функцію згладжування:
   $f(t) = 6t^5 - 15t^4 + 10t^3$
   Ця функція забезпечує плавний перехід між значеннями.

6. **Тривимірна інтерполяція.**
   За допомогою лінійної інтерполяції по всіх трьох координатах обчислюється фінальне значення шуму:
   $\mathrm{lerp}(\mathrm{lerp}(\mathrm{lerp}(S_0, S_1, u), \mathrm{lerp}(S_2, S_3, u), v), \dots)$

У цьому розділі ми розглянемо методи генерації об'єктів для віртуального середовища з використанням концепцій дискретної математики. Ми застосуємо шум Сімплекса (Simplex noise) для початкової генерації та клітинні автомати (Cellular Automata) для покращення розподілу об'єктів.

Шум Сімплекса

Математична Основа

Шум Сімплекса є алгоритмом процедурної генерації, який створює когерентний псевдовипадковий шум. На відміну від шуму Перліна, шум Сімплекса використовує n-вимірний симплекс (узагальнення трикутника), що робить його обчислювально ефективнішим. У нашому проєкті він використовується для генерації об'єктів.

Як видно на рис. 4 (верхній лівий кут), шум Сімплекса створює плавні переходи та природні візерунки, що ідеально підходять для моделювання особливостей ландшафту.

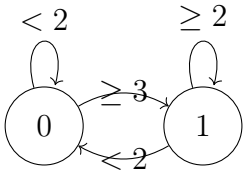
Основна концепція шуму Сімплекса полягає у використанні градієнтних векторів у вершинах n-вимірної прямокутної решітки. Математично це можна виразити так:

$$S(x, y, z) = \sum_{i=1}^{n} \text{grad}_i \cdot \text{dist}_i \cdot \text{fade}(\text{dist}_i) \tag{1}$$

де grad_i — градієнтний вектор, dist_i — відстань до i-ї вершини симплекса, а fade — функція згладжування.

Клітинні автомати

Клітинний автомат — це дискретна модель, що складається з регулярної решітки клітин, кожна з яких може перебувати в одному з скінченної кількості станів. Правила переходу між станами визначаються станами сусідніх клітин.



Для нашої моделі генерації об'єктів ми використовуємо такі правила:

- Народження: порожня клітина стає заповненою, якщо ≥ 3 сусідів заповнені
- Виживання: заповнена клітина залишається заповненою, якщо ≥ 2 сусідів заповнені
- Смерть: заповнена клітина стає порожньою, якщо < 2 сусідів заповнені

На рис. 4 видно результат застосування клітинних автоматів до початкового розподілу об'єктів (порівняйте другий ряд зображень із першим).

Реалізація алгоритму генерації об'єктів

Реалізація шуму Сімплекса

Для генерації шуму Сімплекса ми створили клас `SimplexNoise`:

```
class SimplexNoise:
    def __init__(self, seed):
        self.seed = seed
        random.seed(seed)

        self.perm = list(range(256))
        random.shuffle(self.perm)
        self.perm += self.perm

        self.grad3 = [
            (1,1,0), (-1,1,0), (1,-1,0), (-1,-1,0),
            (1,0,1), (-1,0,1), (1,0,-1), (-1,0,-1),
            (0,1,1), (0,-1,1), (0,1,-1), (0,-1,-1)
        ]
```

Лістинг 1: Клас SimplexNoise

Основний метод для обчислення шуму:

```
def noise3d(self, x, y, z):
    X = int(math.floor(x)) & 255
    Y = int(math.floor(y)) & 255
    Z = int(math.floor(z)) & 255

    x -= math.floor(x)
    y -= math.floor(y)
    z -= math.floor(z)

    u = self.fade(x)
    v = self.fade(y)
    w = self.fade(z)

    A = self.perm[X] + Y
```

```
    AA = self.perm[A] + Z
    AB = self.perm[A + 1] + Z
    B = self.perm[X + 1] + Y
    BA = self.perm[B] + Z
    BB = self.perm[B + 1] + Z

    return w * (y2 - y1) + y1
```

Лістинг 2: Метод обчислення 3D шуму

Розміщення об'єктів із використанням шуму

Для визначення, чи можна розташувати об'єкт у певній позиції, ми використовуємо функцію, що поєднує кілька октав шуму:

```
def can_place(coordinates, seed, region=Region.STEPPE,
   intensity=0.03):
    x, y, z = coordinates
    noise = get_simplex_noise(seed)

    scale = 0.05 if region != Region.STEPPE else 0.08

    base_noise = noise.noise3d(x * scale, y * scale, z * scale)
    detail_noise = noise.noise3d(x * scale * 2, y * scale * 2,
       z * scale * 2) * 0.5

    if region == Region.STEPPE:
        micro_detail = noise.noise3d(x * scale * 4, y * scale *
            4, z * scale * 4) * 0.25
        combined_noise = (base_noise + detail_noise +
           micro_detail) * 0.4
    else:
        combined_noise = (base_noise + detail_noise) * 0.5

    normalized_noise = (combined_noise + 1) * 0.5

    threshold = compute_threshold(region, intensity)

    return normalized_noise > threshold
```

Лістинг 3: Функція розміщення об'єктів

Застосування клітинних автоматів

Для покращення розподілу об'єктів ми застосовуємо клітинний автомат, який видаляє ізольовані об'єкти та збільшує щільність у груповому розташуванні:

```
def apply_cellular_automata(object_map, iterations=3,
    birth_threshold=3, survival_threshold=2):
    height, width = object_map.shape
    result = object_map.copy()

    for _ in range(iterations):
        next_gen = result.copy()
        for i in range(1, height-1):
            for j in range(1, width-1):
                neighbors = np.sum(result[i-1:i+2, j-1:j+2]) -
                    result[i, j]

                if result[i, j] == 0 and neighbors >=
                    birth_threshold:
                    next_gen[i, j] = 1
                elif result[i, j] == 1 and neighbors <
                    survival_threshold:
                    next_gen[i, j] = 0

        result = next_gen

    return result
```

Лістинг 4: Реалізація клітинного автомата

Результати візуалізації та аналіз

Вплив клітинних автоматів на розподіл об'єктів

Як видно з візуалізації на рис. 4, застосування клітинних автоматів значно покращує розподіл об'єктів у віртуальному середовищі:

- Ізольовані об'єкти видаляються за рахунок правила виживання (потреба у мінімум 2 сусідах)
- Групи об'єктів розширюються за рахунок правила народження (3+ сусідів)

- Створюються більш природні кластери об'єктів замість випадкового розподілу

Діаграми у нижній частині рис. 4 показують зміну кількості об'єктів до та після застосування клітинних автоматів, а також розподіл типів об'єктів за категоріями.

Висновки

У цьому розділі ми розглянули методи генерації об'єктів для віртуального середовища з використанням дискретної математики:

1. Ми реалізували шум Сімплекса для початкової генерації об'єктів з використанням детермінованої псевдовипадковості
2. Для покращення результатів ми застосували клітинні автомати — скінченні автомати, що працюють з двовимірною решіткою
3. Наша реалізація демонструє ефективність поєднання континуального шуму (Сімплекс) і дискретної моделі (клітинний автомат) для отримання природного розподілу об'єктів
4. Методи візуалізації допомогли нам оцінити результати та налаштувати параметри алгоритмів

Цей підхід можна використовувати для створення процедурно генерованих ландшафтів у відеоіграх, симуляціях та інших інтерактивних середовищах.

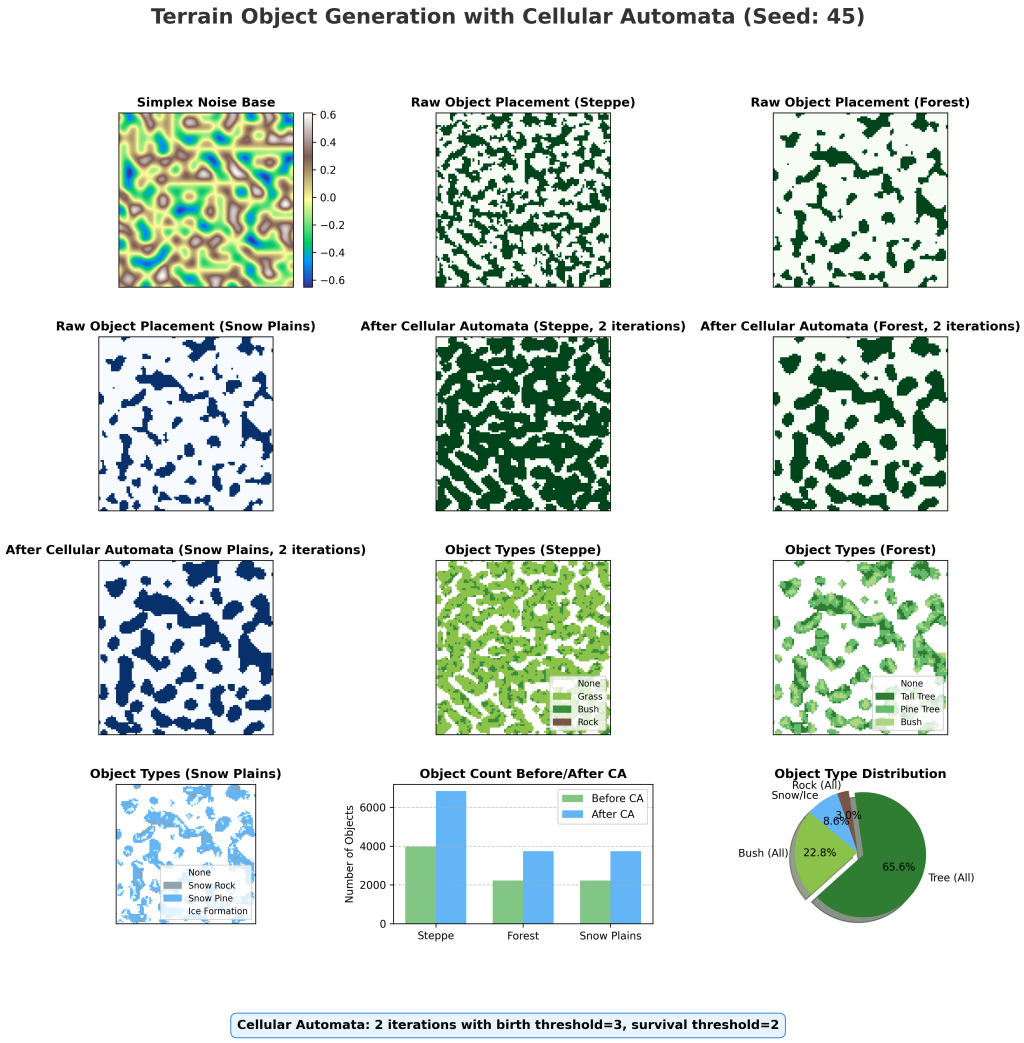


Рис. 4: Візуалізація процесу генерації об'єктів: базовий шум Сімплекса (верхній лівий кут), розміщення об'єктів до та після застосування клітинних автоматів, розподіл типів об'єктів за регіонами