Generating Procedural Tilemaps with Wave Function Collapse and Markov Chains

Tommaso Facchin

February 2025

Abstract

This document presents a procedural tilemap generator based on the Wave Function Collapse (WFC) algorithm and Markov transition matrices. The goal is to create coherent and structured 2D maps that follow predefined adjacency constraints, ensuring a logical and varied tile arrangement. This approach finds applications in game development, simulations, and interactive environments.

1 Introduction

Procedural generation techniques allow for the automatic creation of complex structures without manual intervention. This is widely used in games like *Minecraft*, *No Man's Sky*, and *The Binding of Isaac* to create dynamic, unpredictable worlds.

This project focuses on generating **tile-based maps** using a hybrid approach that combines **Wave Function Collapse (WFC)** with **Markov Chains**. The **WFC algorithm** ensures local consistency, while **Markov transition matrices** introduce global probabilistic constraints, making the generated maps more structured and varied.

2 Wave Function Collapse (WFC)

The **Wave Function Collapse** algorithm is inspired by quantum mechanics but is actually much simpler. The idea is that every tile starts in a **superposition** of all possible tiles, and as constraints are applied, the possibilities collapse until only one tile remains per grid cell.

2.1 Entropy in WFC

To determine which tile should collapse next, WFC uses **entropy**, a measure of uncertainty. We define entropy using the Shannon entropy formula:

$$H(X) = -\sum_i P(x_i) \log P(x_i) \tag{1}$$

where:

- $P(x_i)$ is the probability of choosing tile x_i.
- A higher entropy means more uncertainty, and vice versa.

After a tile is collapsed, its neighbors' entropy values must be updated. The new entropy for a cell C_j is computed as:

$$H(C_j) = -\sum_i P(x_i \mid N_j) \log P(x_i \mid N_j) \tag{2}$$

where $P(x_i \mid N_j)$ is the probability of tile x_i appearing in C_j given its neighbors N_j. The WFC algorithm ensures that entropy never increases:

$$H(C_{j+1}) \leq H(C_j) \tag{3}$$

2.2 Step-by-Step Breakdown

1. **Initialize the Grid:** Each cell starts in a superposition state, meaning it could be any tile.
2. **Compute Entropy:** For each cell, calculate entropy using Equation (1).
3. **Pick the Lowest Entropy Cell:** The tile with the least uncertainty is chosen.
4. **Collapse the Tile:** A tile is chosen based on weighted probabilities.
5. **Propagate Constraints:** The choice affects neighbors, limiting their possibilities.
6. **Repeat Until Fully Collapsed:** The process continues until all cells have a single tile.

3 Markov Chains and Transition Matrices

Markov Chains introduce structured randomness into the tilemap by controlling the probability of tile placement based on prior states.

3.1 Definition of a Markov Chain

A **Markov Chain** is a stochastic process where the probability of transitioning to the next state depends only on the current state:

$$P(X_{n+1} \mid X_n, X_{n-1}, ..., X_0) = P(X_{n+1} \mid X_n) \tag{4}$$

3.2 Transition Matrices

A Markov Chain is represented by a **transition matrix** T, where each element T_{ij} gives the probability of moving from tile s_i to tile s_j:

$$T = \begin{bmatrix} P(G \mid G) & P(W \mid G) & P(S \mid G) \\ P(G \mid W) & P(W \mid W) & P(S \mid W) \\ P(G \mid S) & P(W \mid S) & P(S \mid S) \end{bmatrix} \tag{5}$$

3.3 Steady-State Solution

The **steady-state distribution $\boldsymbol{\pi}$** satisfies:

$$\boldsymbol{\pi} T = \boldsymbol{\pi} \tag{6}$$

Expanding for a **3-tile system**:

$$\pi_G P(G \mid G) + \pi_W P(G \mid W) + \pi_S P(G \mid S) = \pi_G \tag{7}$$

$$\pi_G P(W \mid G) + \pi_W P(W \mid W) + \pi_S P(W \mid S) = \pi_W \tag{8}$$

$$\pi_G P(S \mid G) + \pi_W P(S \mid W) + \pi_S P(S \mid S) = \pi_S \tag{9}$$

Solving this system gives **long-term tile distributions**.

3.4 Higher-Order Markov Chains

Basic Markov Chains only consider the **previous state**. However, we can extend this by using **second-order** models:

$$P(X_{n+1} \mid X_n, X_{n-1}) = P(X_{n+1} \mid X_n) P(X_n \mid X_{n-1}) \tag{10}$$

This allows **larger structures** (like rivers and roads) to emerge naturally.

4 Combining WFC and Markov Chains

To integrate Markov Chains into WFC:

1. **Initialize the grid using WFC.**
2. **Modify entropy using Markov probabilities** (Equation 1).
3. **Collapse cells step by step**, updating probabilities dynamically.
4. **Ensure large-scale consistency** by enforcing transition constraints.

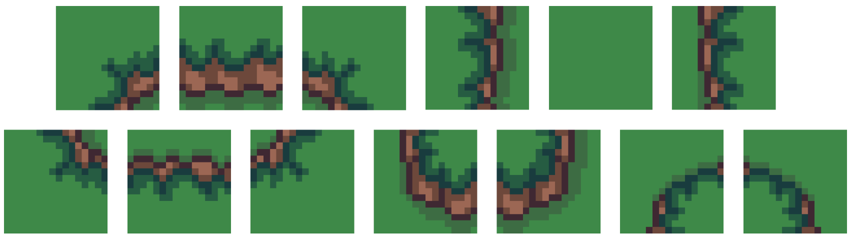

Figure 1: The tileset i used for this project

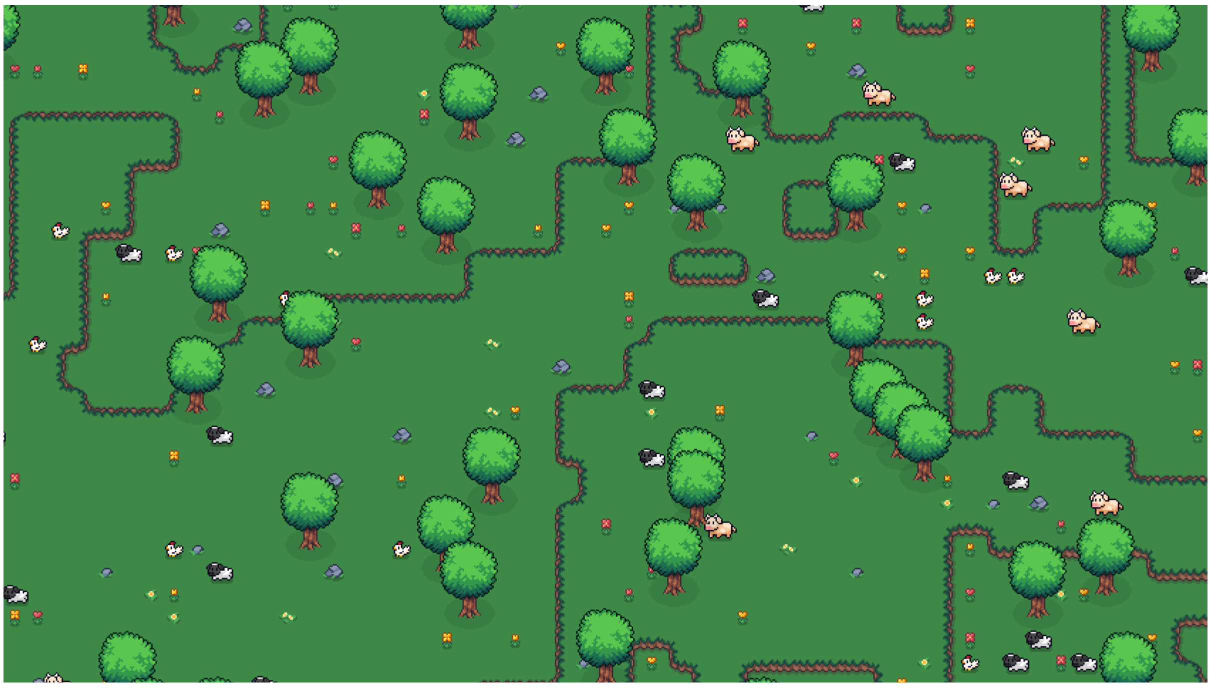

Figure 2: A tilemap using Markov transitions to create structured patterns

5 Code

The following code implements the Markov-Wave Function Collapse (WFC) algorithm to generate procedural tilemaps. It integrates the WFC method with Markov Chains to create locally consistent and globally structured tilemaps.

5.1 Cell Class

The `Cell` class represents a single cell in the Wave Function Collapse grid. Each cell has two properties:

1. `nTile` (of type `int`): The index of the tile assigned to this cell. A value of -1 indicates that the cell has not been collapsed yet.

2. `entropy` (of type `int`): The entropy (or uncertainty) of the cell, representing the number of valid tile options for this cell. A higher value indicates more possible tiles, while a lower value indicates fewer possible tiles.

The `Cell` class has a constructor that initializes these two properties:

1. `Cell(int nTile, int entropy)`: This constructor accepts two parameters: the tile index (`nTile`) and the entropy value (`entropy`) and assigns them to the corresponding properties.

```
public class Cell
{
    public int nTile;
    public int entropy;

    public Cell(int nTile, int entropy)
    {
        this.nTile = nTile;
        this.entropy = entropy;
    }
}
```

5.2 The Init function

The `Init()` function is responsible for setting up the initial environment for the Wave Function Collapse (WFC) process. It performs the following tasks:

1. Initializes a 2D array of `TextMeshProUGUI` elements to display entropy values for each grid cell.
2. Creates a grid of `Cell` objects, each initialized with a default entropy value and an unassigned tile.
3. Instantiates text objects and assigns their positions on the canvas.
4. Initializes the transition matrices for each direction (top, bottom, left, right).
5. Calls the `PopulateMatrix()` function to populate the transition matrices with appropriate values.
6. Starts the collapse process using the `StartCoroutine(CollapseCell())` function, which will begin the wave function collapse.

```
private void Init()
{
    // Initialize text elements
    textElements = new TextMeshProUGUI[dimension, dimension];

    // Initialize grid and matrix
    grid = new Cell[dimension, dimension];

    for (int x = 0; x < dimension; x++)
    {
        for (int y = 0; y < dimension; y++)
        {
            grid[x, y] = new Cell(-1, 0);

            GameObject textObj = Instantiate(textPrefab, canvasTransform);

            textElements[x, y] = textObj.GetComponent<TextMeshProUGUI>();
            textElements[x, y].text = grid[x, y].entropy.ToString();

            textObj.GetComponent<RectTransform>().anchoredPosition = new Vector2(y * 100, -x *
                100);
        }
    }

    // Initialize transition matrix
    transitionMatrixTop = new float[tilesCount, tilesCount];
    transitionMatrixBottom = new float[tilesCount, tilesCount];
    transitionMatrixLeft = new float[tilesCount, tilesCount];
    transitionMatrixRight = new float[tilesCount, tilesCount];

    GetComponent<PopulateMatrix>()._PopulateMatrix();

    // Start collapsing
    StartCoroutine(CollapseCell());
}
```

5.3 CollapseCell function

The `CollapseCell()` function performs the core of the Wave Function Collapse algorithm. It follows these steps:

1. Calls `EvaluateEntropy()` to compute and update the entropy for each cell in the grid.
2. Calls `CollapseLowestEntropy()` to collapse the cell with the lowest entropy, thus reducing uncertainty.
3. Updates the grid with the newly collapsed state by calling `UpdateMatrix()`.
4. Uses `yield return new WaitForSeconds(0.001f);` to create a small delay before the next iteration, allowing for smoother animation.
5. If the grid still has cells with uncertainty, the function calls itself recursively using `StartCoroutine(CollapseCell()`
   If all cells have collapsed, it proceeds to the `Plant()` function to finalize the process.

```
private IEnumerator CollapseCell()
{
    EvaluateEntropy();
    bool doLoop = CollapseLowestEntropy();
    UpdateMatrix();

    yield return new WaitForSeconds(.001f);

    if (doLoop) StartCoroutine(CollapseCell());
    else Plant();
}
```

5.4 EvaluateEntropy function

The `EvaluateEntropy()` function updates the entropy of each grid cell. It works as follows:

1. Iterates over each grid cell, calculating its entropy using the `GetEntropy(x, y, 1)` function.
2. If the entropy for a cell is zero (i.e., it has been fully collapsed), it calls the `Explode(x, y)` function to reset the surrounding cells.
3. After updating the entropy values for all cells, it calls `UpdateMatrix()` to reflect the changes in the grid and update the visual representation.

```
void EvaluateEntropy()
{
    for (int x = 0; x < dimension; x++)
    {
        for (int y = 0; y < dimension; y++)
        {
            grid[x, y].entropy = (int)GetEntropy(x, y, 1);
            if (grid[x, y].entropy == 0) Explode(x, y);
        }
    }
    UpdateMatrix();
}
```

5.5 GetEntropy function

The `GetEntropy()` function calculates the entropy (uncertainty) of a given grid cell, considering the transition probabilities from its neighbors. The function works as follows:

1. If the tile has already been collapsed (i.e., its `nTile` is not -1) and `type` is 1, it returns an entropy of 1.
2. Otherwise, it initializes a probability array and assigns an initial value of 1 for each tile.
3. It then checks the neighboring tiles (top, bottom, left, right) and adjusts the probabilities based on the transition matrices.
4. Depending on the `type` argument:
   - If `type` is 1, it counts the number of valid tile options (where the probability is greater than 0).
   - If `type` is 2, it returns the probability array of valid tile transitions.
5. The final result is either the number of valid options (entropy) or the probability array (for further processing).

```
private object GetEntropy(int x, int y, int type)
{
    if (grid[x, y].nTile != -1 && type == 1) return 1;

    float[] probabilities = new float[tilesCount];
    for (int i = 0; i < probabilities.Length; i++) probabilities[i] = 1;

    //Top
    if (y < dimension - 1 && grid[x, y + 1].nTile != -1)
        for (int i = 0; i < probabilities.Length; i++) probabilities[i] *= transitionMatrixTop[
            grid[x, y + 1].nTile, i];

    //Bottom
    if (y > 0 && grid[x, y - 1].nTile != -1)
        for (int i = 0; i < probabilities.Length; i++) probabilities[i] *= transitionMatrixBottom
            [grid[x, y - 1].nTile, i];

    //Left
    if (x > 0 && grid[x - 1, y].nTile != -1)
        for (int i = 0; i < probabilities.Length; i++) probabilities[i] *= transitionMatrixLeft[
            grid[x - 1, y].nTile, i];

    //Right
    if (x < dimension - 1 && grid[x + 1, y].nTile != -1)
        for (int i = 0; i < probabilities.Length; i++) probabilities[i] *= transitionMatrixRight[
            grid[x + 1, y].nTile, i];

    switch (type){
        case 1:
            int c = 0;
            for (int i = 0; i < probabilities.Length - 1; i++) if (probabilities[i] > 0) c++;
            return c;
        case 2:
            return probabilities;
    }

    return 0;
}
```

5.6 CollapseLowestEntropy function

The `CollapseLowestEntropy` function identifies the cell with the lowest entropy and collapses it. This is done by selecting a tile based on the calculated probabilities and updating the state of the grid.

```
private bool CollapseLowestEntropy()
{
    Vector2Int position = new Vector2Int(-1, -1);
    int minEntropy = int.MaxValue;

    for (int x = 0; x < dimension; x++)
    {
        for (int y = 0; y < dimension; y++)
        {
            if (grid[x, y].entropy < minEntropy && (UnityEngine.Random.Range(0, 2) == 0 ||
                 minEntropy == 100) && grid[x, y].nTile == -1)
            {
                minEntropy = grid[x, y].entropy;
                position = new Vector2Int(x, y);
            }
        }
    }

    if (minEntropy == int.MaxValue) return false;

    float[] probabilities = (float[])GetEntropy(position.x, position.y, 2);
    float[] normalizedProbabilities = new float[tilesCount];
    float sum = 0;

    for (int i = 0; i < probabilities.Length; i++) sum += probabilities[i];
    for (int i = 0; i < probabilities.Length; i++) normalizedProbabilities[i] = probabilities[i]
         / sum;

    float r = UnityEngine.Random.Range(0f, 1f);
    float cumulativeProbability = 0;

    for (int i = 0; i < probabilities.Length; i++)
    {
        cumulativeProbability += normalizedProbabilities[i];
        if (r <= cumulativeProbability)
        {
            grid[position.x, position.y].nTile = i;
            break;
        }
    }

    return true;
}
```

5.7 Explode function

The `Explode` function is used to reset the possible tile options of neighboring cells if a cell reaches a state where no valid tile options remain. This ensures the grid remains consistent.

```
private void Explode(int x, int y)
{
    for (int i = x - 1; i <= x + 1; i++)
    {
        for (int j = y - 1; j <= y + 1; j++)
        {
            if (i >= 0 && i <= dimension - 1 && j >= 0 && j <= dimension - 1)
                grid[i, j].nTile = -1;
        }
    }
}
```

5.8 Plant function

Once the grid is collapsed, the `Plant` function is responsible for planting prefabs in the tilemap based on the generated grid. This function instantiates prefabs based on the tile type in the grid.

```
private void Plant()
{
    for (int x = 0; x < dimension; x++)
    {
        for (int y = 0; y < dimension; y++)
        {
            if (tilemap.GetTile(new Vector3Int(-x, -y, 0)) == tiles[4])
            {
                if (y != dimension - 1 && y != 0 && x != dimension - 1 && x != 0)
                {
                    int random = UnityEngine.Random.Range(0, 100);
                    if (random < Prefabs.Length) Instantiate(Prefabs[random], new Vector3(-x +
                         tilemap.transform.position.x, -y + tilemap.transform.position.y, dimension
                          - y), Quaternion.identity);
                }
            }
        }
    }
}
```

6 Conclusion and Future Work

This approach integrates the **Wave Function Collapse** (WFC) algorithm with **Markov Chains**, providing a solution for generating tilemaps that are both:

- **Locally consistent**, ensuring adjacency rules are followed (via WFC).
- **Globally structured**, with tile transitions based on probabilistic Markov models.

The resulting tilemaps exhibit meaningful and coherent structures, making them suitable for applications such as procedural content generation in games, where maintaining consistency and diversity is crucial.

If you'd like to explore the implementation of this approach,
you can watch the demonstration video here: Watch the video on YouTube.
or the source code here: Github source code

Future work could enhance this approach by:

- Exploring **higher-order Markov models** for more complex terrain generation.
- Using **machine learning techniques**, such as **recurrent neural networks (RNNs)**, to dynamically adjust transition probabilities based on environmental factors.
- Incorporating **biased transition matrices** to control the distribution of specific biomes or regions.
- Optimizing the computational efficiency to handle larger and more complex environments.
- Extending this methodology to 3D spaces or real-time procedural generation, offering new possibilities for dynamic content creation.

Ultimately, the integration of Markov Chains into WFC improves the algorithm's ability to capture both local and global patterns, leading to more visually compelling and consistent results. The continued development of this approach could provide powerful tools for procedural content generation in diverse applications.