Exploring Procedural Generation Algorithms in Unity 2D

Ryan Overbeck

April 23, 2025

Contents

1 Introduction

TODO - include link to repo here

2 Binary Space Partitioning Approach

> *"A binary space partition takes a given space and splits it in half, and then takes the two areas that were created and splits those in half, and repeats until some threshold is reached."* (Short and Adams, 2017, p. 293)

This quote sums up our first approach to procedurally generate rooms for our 2D dungeon. In this implementation, we define a rectangular region—the entire dungeon area—and iteratively subdivide it into smaller rectangles (potential "rooms") until no further meaningful subdivision is possible Lastly, some offset is applied to create space between these rooms. The following outlines how this simplified Binary Space Partitioning (BSP) approach is handled in our Unity project.

2.1 Setting up the Initial Partition

The process begins by defining a single large bounding area using a `BoundsInt` structure. As noted in the Unity documentation, the `BoundsInt` structure "represents an axis aligned bounding box with all values as integers" (Unity Technologies). This is useful for several reasons, among them since `BoundsInt` uses integer coordinates, it aligns naturally with tile-based systems (where tiles or grid cells are also defined by integral x, y positions).

Next, this bounding area is enqueued in a `Queue<BoundsInt>` which manages partitions that still need to be processed. A queue-based approach was selected to keep the partitioning logic simple and easy to follow. A stack-based approach would have worked equally as well—only changing the order of processing.

2.2 Iterative Subdivision

While there are partitions in the queue:

- Dequeue the next partition.
- Check if the current partition meets or exceeds the *minimum width* (`minWidth`) and *minimum height* (`minHeight`) constraints.
- If it can be subdivided, choose randomly whether to *prefer a horizontal* or *vertical* split (a 50/50 chance). Attempt to partition the space along that orientation if there is sufficient size to do so at least twice (e.g., if the height is at least `2 * minHeight` for a horizontal split).
- If the chosen orientation is not feasible, try the alternative orientation.
- If neither orientation is possible (the partition is too small), mark that partition as finalized (a "room") and add it to the `partitions` list.

2.3 Managing Subdivisions

The actual subdivision is performed in two helper methods, `PartitionHorizontally` and `PartitionVertically`. These methods randomly determine a *split point* within the valid range and create two new `BoundsInt` objects representing the subdivided spaces. Each new partition is enqueued for further subdivision if possible.

- *Horizontal Partition:* Splits along the y-axis, producing a top and bottom rectangle.
- *Vertical Partition:* Splits along the x-axis, producing a left and right rectangle.

By alternating or randomly selecting horizontal versus vertical splits, the algorithm produces layouts of different shapes, preventing overly predictable results.

2.4 Applying Offset

TODO

3 Random Walk Approach

> *"Random walks are produced by starting at a given point and then taking steps in random directions. Many natural processes, like molecular motion, can be described by random walks, and they can be useful for generating all sorts of naturalistic paths and features."* (Short and Adams, 2017, p. 286)

This quote sums up our second approach to procedurally generate rooms for our 2D dungeon. In this implementation, we choose a starting position and perform a defined number of "walks," each consisting of a certain number of steps. Every time we move to a new position, we record that position in a set of coordinates. After all walks are completed, these coordinates form the layout of our dungeon floor. The following outlines how this is handled in our Unity project.

3.1 Initial Conditions

A `HashSet` is initialized to keep track of every visited position (avoiding duplicates).

3.2 Iterations (Walks)

The number of iterations (walks) is determined by an input parameter. For each walk:

- A `Vector2Int` for storing the previous position is set to the start position (provided by an input parameter).
- Perform a fixed number of steps, where each step is taken in a randomly chosen direction (up, down, left, or right). The new position after each step is added to the set, and the previous position is updated.

3.3 Results

After all walks and steps are completed, the algorithm returns the set of all unique positions visited during the random walk. This collection of positions is used to place the floor tiles. The resulting dungeons will tend to have a more organic winding layout, in contrast to those resulting from the binary space partitioning (BSP) approach with its more methodical grid-like arrangement.

4 Wave Function Collapse Approach

The Wave Function Collapse (WFC) algorithm, specifically the Simple Tiled variant is a procedural generation method that draws inspiration from quantum mechanics to create tile-based patterns. It works by assigning a set of potential states to each tile and then progressively reducing these possibilities based on constraints from adjacent tiles, much like the collapse of a quantum state into a definite outcome.

A simplified version of this algorithm forms the third approach for generating the layout of our dungeon. Its implementation in our Unity project is explored below.

4.1 Setting up

4.1.1 Adjacency Constraints

The first step in setting up the algorithm is to establish valid tile adjacencies for the supplied tileset. To ensure the tiles tile appropriately, every pixel on the edge of neighboring tiles must be the same color (see figure 1). To that end, the method `GetTilesEdgesColors` accepts a list of tiles which are of the type `Texture2D` (representing bitmaps), it then extracts and returns the color information for every pixel on each tile's edge.

4.1.2 Finding Viable Tiles for each Tile's Edge

The next step is to determine which tiles satisfy the adjacency constraints of every tile. The method `GetTilesEdgesViableTiles` performs this task by iterating through each tile and examining every one of its four edges. For each edge, the method identifies its complementary edge (i.e., the bottom edge is compared with the top edge, and the left edge with the right edge) using a helper function `GetComplementaryEdge`.

The method then compares the colors of the current edge with the corresponding complementary edge of every tile by using a pixel-by-pixel comparison via the `EdgesMatch` helper function. If the two edges match exactly, the index of the candidate tile is recorded as a viable neighbor. This process results in a three-dimensional array where each element contains the list of tile indices that can be placed adjacent to a given tile on a specific edge.

This step is crucial as it ensures that only tiles with matching edge colors are considered for adjacent placement, thus maintaining the seamless appearance of the tiled pattern.

4.1.3 Mapping Tiles to Weights

The function `GetTilesToWeights` is responsible for constructing a dictionary that maps each tile's identifier (an integer index) to its corresponding weight. These weights, provided in the `tileWeights` list, can be interpreted as the desirability of selecting a given

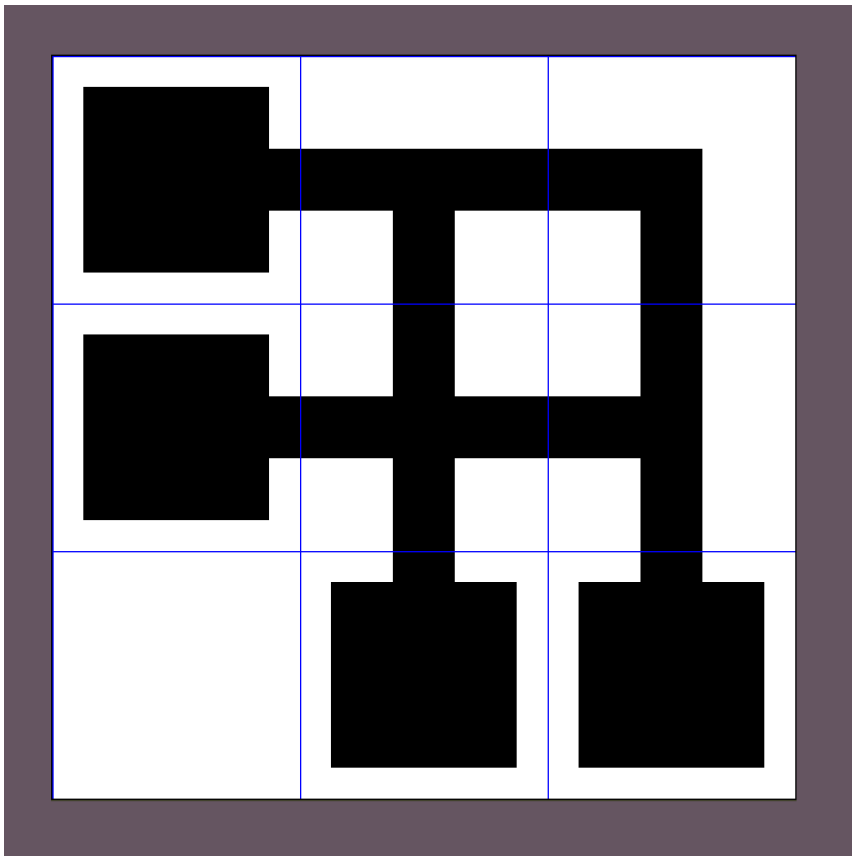

Figure 1: Tile Adjacency Constraints

tile during the cell collapse process. By iterating over the total number of tiles, the function assembles a lookup table that will be used by each cell to initialize its set of viable tiles.

4.1.4 Cell Structure and Entropy Calculation

The `Cell` class encapsulates the state of each grid cell. Each cell maintains a dictionary (`viableTilesToWeights`) that reflects the current set of possible tiles along with their weights. Crucially, the cell also stores a computed value called `entropy`. In this context, entropy is a measure of uncertainty or randomness over the set of viable options, computed using an approach similar to the Shannon entropy formula.

4.1.5 Grid Initialization

The function `InitializeOutputGrid` is tasked with constructing the overall grid that represents the output of the procedural process. The grid is implemented as a dictionary where each key is a two-dimensional coordinate (represented by a `Vector2Int`) and the value is an instance of the `Cell` class.

4.2 The Algorithm

The algorithm is driven by the `CollapseCells` method which for every cell in the grid:

1. Collapses the minimum-entropy cell.

2. Enqueues that cell for neighbor-constraint propagation.

3. Dequeues cells one by one, and for each of the four cardinal neighbors:

    - Skips neighbors outside the grid or already collapsed.
    - Builds the set of *allowed* neighbor tiles by looking up, for each currently viable tile in the source cell, which adjacent tiles are compatible on the shared edge (using `tilesEdgesViableTiles[tile][edge]`).
    - Prunes any tiles from the neighbor's `viableTilesToWeights` that are not in this allowed set.
    - If the neighbor's viable set becomes empty, returns `true` to signal a contradiction.
    - Otherwise, if pruning reduced its size, enqueues the neighbor for further propagation.

By iteratively collapsing low-entropy cells and propagating adjacency constraints, this method enforces global consistency across the grid or detects unsatisfiable configurations. If a contradiction is found, the grid is re-initialized and the process repeats.

4.3 Extracting Tilemap Positions

The function `GetPositionsFromGrid` takes the fully collapsed `outputGrid` and the list of source tile textures, and produces a `Queue<Vector2Int>` like the other position generators. It does so by iterating over every pixel within each cell's tile (`Texture2D`): if the sampled `Color` has red, green, and blue channels all equal to zero (i.e. pure black), the method computes the absolute tilemap position by scaling the cell's grid coordinate by the tile dimensions and adding the pixel offset.

5 Difficulties

5.1 Challenges in Adapting WFC to Unity

The Wave Function Collapse (WFC) algorithm offers an innovative approach to procedural content generation. However, adapting it to a Unity 2D project presented several challenges, primarily due to shortcomings in the primary reference, the source for it on GitHub. Gumin

A major source of difficulty was the repository's README, which conflates two distinct variants of the algorithm: *Simple Tiled* and *Overlapping*. The failure to clearly differentiate between these variants led to confusion in the description of the algorithm which was crucial for reasoning about the source code due to poor naming conventions, lack of inline documentation, and terminology borrowed from Quantum Mechanics despite a tenuous connection.

This was not a unique difficulty, as one researcher noted, "in personal correspondence with several users of WFC, we learned that many of them treated the code as a black box". (Karth and Smith, 2017)

5.2 Challenges in Moving to Tile Generation Over Time

When the project first aimed to generate a dungeon floor, the tile placement occurred synchronously on `Start()`. This approach was straightforward to implement but lacked any concept of observable progression over time. Addressing this necessitated several changes.

First, the original data structure for storing the positions of the floor tiles, a `HashSet` needed to be changed so that order could be preserved. Using a `Queue` was necessary to maintain first-in-first-out order. However, this change introduced a new challenge: since a `Queue` alone does not handle duplicates, a separate `HashSet` had to be used in parallel to record visited positions. By checking against this `HashSet` before enqueueing a new coordinate, the system could retain an ordered structure while still preventing repeated tiles from being added. Note that while Unity's `Tilemap.SetTile` method can handle duplicates it would obfuscate how the room is constructed over time so it became necessary to deal with them here.

Second, switching from a synchronous approach to an incremental, time-based approach introduced the need for coroutines. Previously, the entire tile set could be drawn in a single loop, effectively blocking until all positions were placed. In contrast, placing one tile every half-second requires the code to yield partway through execution so that the game loop can continue running. Implementing the coroutine `DrawFloorTilesOverTime` allowed each tile placement to be spaced out over time without freezing the entire game.

Third, as tiles were placed over time, it became important to communicate the dungeon generation's progress to the player. A simple UI element was introduced to display both the total number of tiles to be placed and the number of tiles already placed, updating in real time as new positions were set.

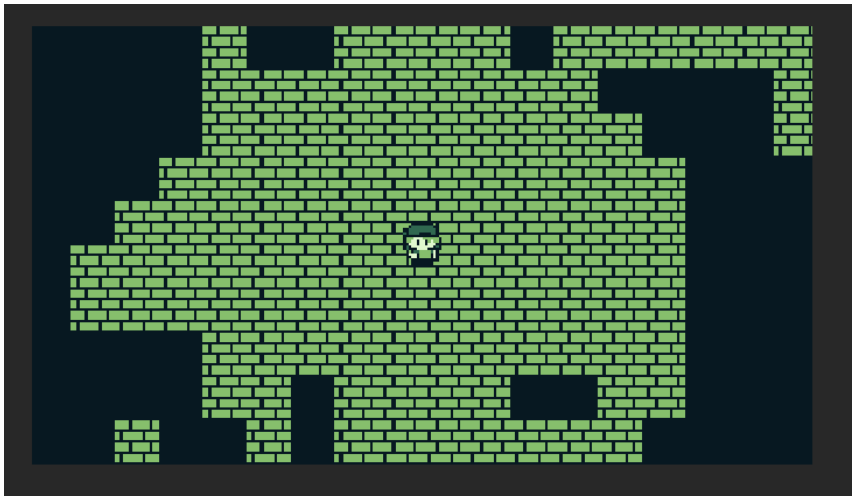

Figure 2: Challenge 4.1 - game view before

5.3 Challenges in implementing WFC

TODO: Describe debugging process: Issue with selecting minimum entropy cell, always collapsing first cell due to no isCollapsed flag. Issue with not flipping Y on OutputGrid to match Unity's coordinate system. Issue with only propagating adjacency constraints from collapsed cell to 4-cardinal neighbors instead of having it ripple out to every cell with changed possibilities.

Figure 3: Challenge 4.1 - game view after

6 Impact of Previous Courses on Project

6.1 CSCI 451 and Utilizing Unity Coroutines

Threads in C# represent an operating-system-level construct. As stated by Silberschatz *et al.*, "A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack" Silberschatz et al. (2007). This means that each thread is scheduled independently by the operating system and can take advantage of multiple processor cores for true parallelism. However, the creation and management of multiple threads can be resource-intensive, and programmers must carefully manage data shared between threads to avoid race conditions.

Unity coroutines, on the other hand, are not scheduled by the operating system. They are functions "that can be suspended, resumed, and executed cooperatively (often on the same thread)" Nosenko. In Unity, coroutines primarily run on the main thread and yield control voluntarily, allowing the engine to manage when to resume them. This cooperative model avoids the complexity of thread synchronization, but it does not offer true parallelism by default. Coroutines are particularly useful for tasks that must be broken into incremental steps across frames, such as animations or in the case of this project, long-running operations where you do not want to block the main thread.

My understanding of these concurrency mechanisms was greatly enhanced by taking *CSCI 451 Operating Systems*. The course covered core concepts of processes, threads, synchronization, and CPU scheduling, providing the theoretical background to appreciate how the operating system manages threads and why threads are relatively expensive compared to language-level, cooperative concurrency mechanisms like coroutines. By exploring concurrency at the OS level first, it enables deeper insight into how coroutines abstract these complexities and thus offer a simpler model for many game development scenarios.

6.2 WRIT 102 and Academic Sourcing/Technical Writing

In addition to technical courses, the project benefited significantly from the skills acquired from University Studies, such as *WRIT 102 Introduction to Academic Writing*. This course, which emphasized careful analysis, systematic research, and the construc-

tion of coherent academic arguments laid a solid groundwork for both identifying credible resources and organizing the paper effectively.

The techniques learned in *WRIT 102* proved instrumental during the research phase by facilitating the systematic identification and evaluation of academic sources that supported the project. Moreover, the course's focus on proper documentation and citation ensured that all referenced materials were integrated seamlessly and accurately, thereby maintaining the academic integrity of the paper.

Lastly, the critical reading and persuasive writing strategies honed in the course helped in articulating complex technical concepts clearly and persuasively, bridging the gap between technical content and academic presentation.

7 Conclusion

References

M. Gumin. Wave function collapse. `https://github.com/mxgmn/WaveFunctionCollapse/tree/master`. Accessed: 24 March 2025.

I. Karth and A. M. Smith. Wavefunctioncollapse is constraint solving in the wild. *FDG '17: Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017.

A. Nosenko. Asynchronous courotines with c#. `https://learn.microsoft.com/en-us/shows/dotnetconf-2020/asynchronous-courotines-with-c`. Accessed: 28 February 2025.

T. Short and T. Adams. *Procedural Generation in Game Design*. CRC Press, 2017.

A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts with Java*. Wiley, 7th edition, 2007.

Unity Technologies. Unity documentation. `https://docs.unity3d.com/`. Accessed: 10 February 2025.