Project Paper

Ryan Overbeck

February 17, 2025

Abstract

Add abstract here

1 Introduction

Add introduction 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.

References

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

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