Procedurally Generated Adventure Game

Author
dragos-ionut.marcu@s.unibuc.ro

Abstract

This paper aims to cover the programming techniques, asset creation methods, and existing libraries used to make a 2D game that combines gameplay elements from various existing titles, with a focus on level generation.

1 Introduction

The project consists of a 2D Top-Down Roguelike game created using Unity. The player has to progress through procedurally generated levels using various combinations of spells (that have to be found throughout the game) to defeat a certain number of enemies to be able to enter the next level. The game is heavily modularized, allowing for easy scaling if needed.

2 Related Work

The game's progression system is typical of a Roguelike, similar to *The Binding of Isaac*(2011), the player getting stronger and obtaining more powerful equipment as he progresses, but having to start over if he loses once. The game's combat mechanic for the player consists of creating combinations from a limited number of basic spells, inspired by *Magicka 2*(2015). The level layout generation is inspired by the one seen in *Darkest Dungeon*(2015), with the difference that the level's graph of rooms may contain cycles.

3 Method

3.1 Game mechanics

Once the game has started, the player has access to the following actions:

- Moving around the map using the WASD keys and changing the direction faced by the player's character using the mouse
- Opening the inventory using the I or TAB keys, equipping or unequipping armor, wands (both of which modify the player's attributes) and spell books that determine what elements the player can use.
- Using between one and three elements to create spells. The player can select the elements using the 1, 2 and 3 keys. After selecing the desired combination, the player can release the spell with the left mouse button.
- Opening the skills tab using the G key. From here the player can spend skill points, earned through defeating enemies, to upgrade the following attributes: attack damage, maximum health points, maximum mana points, and mana recharge rate.
- Pressing E when facing chests found throughout the map to open them, revealing an item randomly chosen from the chest's loot table that the player can take by selecting an empty slot in their inventory. The E key is also used to open the portal to the next level if the progression conditions are met.

3.2 Game Progression

As of writing this paper, the game progression consists of completing as many levels as possible. The player can go to the next level if two thirds of the enemies have been eliminated and the boss has been defeated, if he exists. The player earns skill points with each enemy defeated which he can use to increase his attributes in the skills tab. Chests are scattered throughout the map which the player can open to potentially obtain stronger equipment or new spells.

3.3 Sprites and Animations

The sprites were either created using *Paint.NET* (dotPDN LLC, version 5.1.6) or were downloaded for free from *OpenGameArt.org* (OpenGameArt, 2025). The sprites are supposed to give the impression of a true bird's-eye perspective, the characters and objects being seen from above by the player. There are two types of animation used in the game, sprite-based and skeletal. Skeletal animations were used for characters that can rotate, such as the player's character, and sprite sheets were used for characters with simpler movement such as the slime.

3.4 Level Generation

The level generation process consists of six steps, in this order: generating the map layout, generating the rooms, generating the corridors, generating the unwalkable tiles, generating the obstacles, and finally generating the entities. The map is based on 2×2 tiles.

The first step does not create anything in the scene yet, but fills a $n \times n$ matrix `roomGrid` (where n) represents the maximum number of rooms per dimension) with ones, indicating where rooms will be placed. The room grid is composed of tiles of size `MaxRoomSize` plus `RoomPadding`, constants currently set to 9 and 4, respectively. The matrix is generated using a Random Walk algorithm, a starting tile position, an end tile position, and the number of rooms (ranging from 8 to 14) are chosen randomly. The initial room is set as the starting room, then all of its neighbors which are within the grid's bounds and have not been selected yet are added to an array `roomCandidates`. A candidate room is chosen at random; then these steps are repeated until the previously set amount of rooms is reached.

The second step generates the rooms and their shape, placing the ground prefabs in the scene. To make the map look more dynamic, Perlin noise is used to create jagged edges for the rooms, a random seed between 0 and 1000 being chosen for both the x and y coordinates that will be used for the noise function. The room length and width are also randomly chosen from the range `MinRoomSize` to `MaxRoomSize`, which are, by default, set to 6 and 9, respectively. The rooms are then positioned in the middle of the first `MaxRoomSize` tiles and their edges have tiles removed by using Perlin noise with the previously mentioned seeds and a noise scale of 0.8 (this low value was chosen to simulate smoother noise). If the Perlin noise function returns a value under 0.45 the tile is removed from the edge, this threshold was chosen after testing multiple values and seeing which one produced the most balanced results. The level generation script uses a matrix `tileGrid` with dimensions $m \times m$ where m is equal to $\texttt{MaxRoomsPerDimension} \times (\texttt{MaxRoomSize} + \texttt{RoomPadding})$; this matrix stores information about each tile, by the end of the second step it's filled with zeros which represent an empty tile and ones which represent a ground tile.

During the third step rooms are connected by generating corridors. The process starts by randomly selecting a corridor candidate from the starting room, then continues from the most recently connected room. Corridors that would form cycles in the room graph are avoided until a spanning tree (a room tree) is established. Afterward, a random number between 0 and 1 determines how many of the remaining unused corridor candidates will be added back to the graph, potentially forming cycles (sometimes none are added). To generate each corridor, a random tile is selected from the facing edges of the two rooms being connected. Straight lines are drawn from these points, meeting at a random position inside the padded area between rooms. During this generation, ground prefabs are placed in the scene and their corresponding tiles are marked with a 4 in the `tileGrid` matrix, representing a tile that is essential for a corridor to be passable. The generation begins one tile before the selected edge tiles in the direction of the corridor. This ensures that Perlin noise-induced modifications do not block the path.

At this point, the borders are ready to be added, the fourth step representing the generation of unwalkable tiles. Each possible tile is checked, if its value within `tileGrid` is 0 and it's within a used room tile or in an empty room tile that borders an existing room, then a border prefab is chosen and the tile is created in the scene. Additionally, if a room is at the edge of the `roomGrid` matrix, an additional border is created, so that the player's camera does not see "empty space" while in-game.

The fifth step is where the obstacles are generated, using Perlin noise again with a randomly generated seed, but this time with a scale value of 2.5 to simulate higher-frequency noise and a threshold of 0.6 to make obstables rarer. The script iterates through every room, calling the Perlin noise function for each tile found with a value of one in the Tile Grid. If the Perlin noise function value is high enough, the tile is marked with a value of 3 in the grid and the eight neighboring tiles are marked with a value of 2, signaling that other obstacles cannot be placed there; this is done to prevent areas from being locked. Another measure taken to prevent locked areas is to not place objects if two or more unwalkable (0) tiles are neighboring the current tile, because that means there is a potential entrance to a one-tile-way there; an obstacle being placed here doesn't guarantee the creation of a locked portion of a room, but this is the most efficient way of assuring locked regions don't appear.

The sixth and final step is where the player, the enemies, the chests and the portal are generated. During the obstacle generation step, if a tile ended up with a value of 1 or 2 it was added to a List of free tiles for this current room. The algorithm chooses a random number of chests that are distributed throughout the rooms, then traverses every room and follows these rules:

- Place the assigned number of chests on free tiles
- If this is the end room, choose a random free tile and create the portal
- If this is the start room, choose a random free tile and spawn the player there
- If this isn't the start room:
  - Choose a random number of enemies to be created (between 0 and $\left\lceil \frac{MaxRoomSize}{2} \right\rceil$).
  - Choose a corresponding amount of free tiles to spawn the enemies.
  - Additionally, if this is the end room choose a free tile where the Boss type enemy should spawn.

Each time a free tile is chosen it is removed from the room's free tile list.

The overall complexity of this entire process is $\mathcal{O}(T^2)$, where $T = MaxRoomsPerDimension \times (MaxRoomSize + RoomPadding)$. As T is less than 200, this complexity is acceptable for a method that's called only when changing the level. Generating the map takes on average 2 seconds, which is due to the amount of creations and deletions of GameObjects. This can be easily improved by implementing object pooling and creating a static background to replace its current generation during the border creation process.

3.5 Enemy AI

The game contains two primary types of enemies: ranged and melee; the former launches attacks when it's within a certain range of the player and has a clear view towards them (A Circle Cast is used inside the Update method when the player is within the enemy's attack range to determine if an attack will reach them), the latter simply damages the player when they're close enough. The game utilizes the Grid Graph implementation of the A* algorithm provided by the A* Pathfinding Project library (Granberg, 2024). Using the A* algorithm, the enemies efficiently move towards the player's character when it enters their range of detection (which can be different from the range of attacking). Melee enemies continue walking until they're right next to the player, then they perform their attack. Ranged enemies walk towards the player's character until it is within their attacking range and there are no objects between them that would stop attacks. These default interactions aren't enabled for Boss type enemies, as they have a separate script controlling their movement. Currently there is only one Boss enemy, so the there is also only one special movement script `RandomPatrolScript`, which randomly selects a free tile from the current room and makes the enemy walk there. If the Boss enemy encounters another enemy on its path, it stops and changes its destination randomly.

The Grid Graph uses eight connections, allowing enemies to move diagonally. The collision testing diameter is reduced from the default value of 1 to 0.9, preventing nearby walkable tiles from being incorrectly marked as unwalkable due to proximity to obstacles. For the A* Pathfinding component within the enemy prefabs, most of the attributes are unmodified, the default values being suitable for 2D Top-Down games. Movement speed and state are managed separately through the EnemyController script. While the A* algorithm handles navigation, additional behavior such as knockback effects caused by attacks require custom handling beyond the default pathfinding capabilities.

4 Future Work

The map generation and enemy movement parts of this project are complete, with only additional content being needed to further improve the game (i.e. more textures and prefabs for new levels with different designs). Important features that still need to be implemented are player death handling, more spell combinations and a start menu and pause menu. Additionally, sound effects, particles for the attack sprites and background music could be added to enhance the overall player experience

5 Conclusion

This project taught me the importance of planning when it comes to game development, as there were multiple instances where I had to rewrite entire systems to make them fit new ones. A good example of this would be needing to rewrite how attack damage was calculated once armor that changed attributes was added, making a lot of the former code obsolete.

Limitations

The main limitation encountered during development was sprite creation, as suitable textures for this style of 2D Top-Down game are difficult to find. As a result, the game's visual style lacks consistency, with textures coming from different sources or being created manually.

References

Aron Granberg. 2024. A* pathfinding project. https://arongranberg.com/astar/. Accessed: 2025.

OpenGameArt. 2025. Opengameart.org - free game assets. https://opengameart.org. Accessed: 2025.