Game Related

General

GameManager is the main entry point of the game, it executes all other scripts. The whole game happens in one scene and is mainly driven through spawning prefabs from the *GameManager* and filling them with data saved as *ScriptableObject*s - the *GameManager* starts by spawning the main menu (with the *UIOverlayManager* attached, which manages most in-game UI overlays - ex. pause menu, game over screen), from which the game is started by pressing a button.
The *GameManager* starts the game by spawning a *LevelManager* from a prefab supplied with the appropriate data from a *LevelSO* scriptable object which then handles spawning the individual rooms.
The *LevelManager* is mainly responsible for spawning rooms - either from a prefab (*SpecialRoom*) or generated with a *MapGenerator*, filling the rooms with enemies (the spawn points and types of which are decided by the *AIGenerator*) and placing the player at the entrance to the room.

Game Entities

Characters (Enemy, Player, Merchant) have *Abilities* (Attack, Dash, Trap, etc.), each *Character* and *Ability* have their own data saved as a *ScriptableObject*

Hierarchy

- Character
    - CombatCharater (IDamageable)
        - EnemyCharacter
        - PlayerCharacter (IPushable)
    - MerchantCharacter
- Ability
    - Attack
        - MeleeAttack
        - RangedAttack
            - AimedRangedAttack
            - ClickRangedAttack
    - Dash
    - Trap

AI

The AI uses behaviour trees - folder “AI/BehaviourTreeBase” contains general and game agnostic behaviour tree code, folder “AI/Trees” contains all character trees, folder “AI/Tasks” all possible tasks.

The game agnostic part consists of a base tree class (*TreeBase*) which stores the reference to the root node and is responsible for updating the tree - i.e. traversing the entire tree every

tick. The tree can contain nodes of several types, each node stores its current status (Running, Success or Failure) and has an Update method which is called when traversing the tree, and returns the current status.

There are several different types of nodes (the internal nodes are all specified in the *NodeBase.cs* file, the leaf nodes each have their own file in the “AI/Tasks” folder), the ones used in the game are as following:

- **Sequence (AND)** - a node with several children which processes all of them in order and reports success only when all of the children have been processed successfully (useful when the enemy can perform a task only provided some conditions are met first - e.g. it can only attack the player if the player is in range, see the attack subtree in Figure 2 for an example)
- **Selector (OR)** - a node with several children which processes all of them in order until a child reports “Success” (used in the game typically to select the first subtree behaviour which it can perform - for instance if an enemy sees the player it can try to perform a melee attack if the player is in range, however if the player is out of range the melee attack will return failure, so the enemy can fall back on a ranged attack instead, see the tree for Ms. Cups in Figure 6 for an example)
- **Sequence with cached last child (THEN)** - similar to the sequence node except the update method doesn’t update the children from the beginning of the list each time but instead picks up in the last child which reported the “Running” state - useful when several behaviours need to be completed in a sequence one after another (for instance the *WalkToTarget* and *WaitFor* tasks in the Patrol subtree - Figure 1)
- **Inverter (NOT)** - a node which simply inverts the result of its child node (used for instance to check if an ability is off cooldown, by inverting the result of the *AbilityOnCD* check)
- **Leaf** - the leaf nodes contain all of the logic of the character behaviours, they can be further divided into:
    - **Tasks** - nodes which instruct the character to perform a behaviour, they succeed after the behaviour has been fully completed (for instance *WalkToTarget* which attempts to move the character to a specified target, or *AttackTarget* which attempts to attack a specified target)
    - **Checks** - nodes which check the state of the game - usually answer a question the AI is asking with a yes (Success) - no (Failure) answer (for instance the *AbilityOnCD* check, which checks whether an ability is on cooldwon)

The characters in the game each have their own behavioral tree but since there are a lot of behaviours which are similar across all enemies, there are some subtrees which are reused and most characters are pieced together from a few different subtrees and a few nodes. The subtrees are in the form of functions which return the root node of the subtree required subtree (all of which are in the *CharacterTreeBase* class) - for instance, the Patrol subtree (illustrated in Figure 1) returns a subtree with a patrolling behaviour which is used as a part of most enemy behaviour trees.

The shared data for each behaviour tree is stored in the tree, it is defined in the *CharacterTreeBase* class. This is useful for instance when setting the target in some tasks -

for example, trying to attack the player requires the enemy to first look for the player inside the level - this is done in *FindTargetInRange* which, provided it finds the player within the specified range, saves the player's position as the target to the shared data, so that the *AttackTarget* task in the tree can access it further down the tree traversal.

Following are some examples of trees and subtrees from the game. The internal nodes are illustrated by a set of keywords for the sake of simplicity:

- Sequence = AND
- Selector = OR
- Sequence with cached last child = THEN
- Inverter = NOT

The patrol subtree consists of a *SequenceWithCachedLastChild* of two alternating tasks - *WalkToTarget* and *WaitFor* which results in the character following a pre-set route.

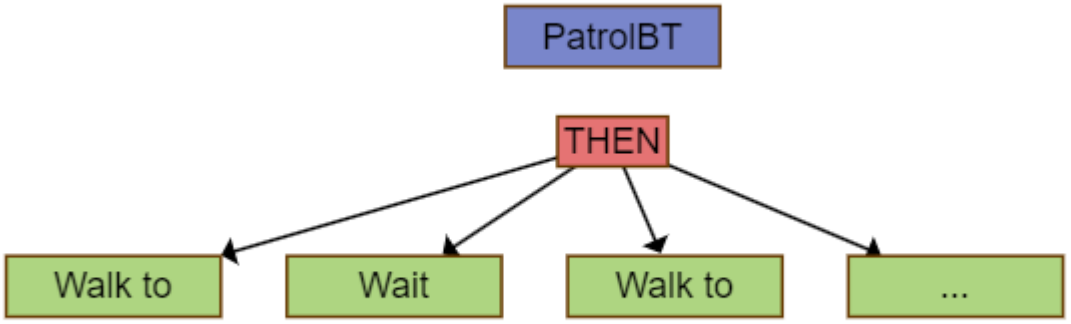

Figure 1: Patrol subtree

The attack subtree returns a subtree for any attack, be it melee or ranged, and optionally checks whether the attack is on cooldown. The *WaitFor* task at the end of the attack serves as attack recovery (the attack recovery time is set in the attack's data).

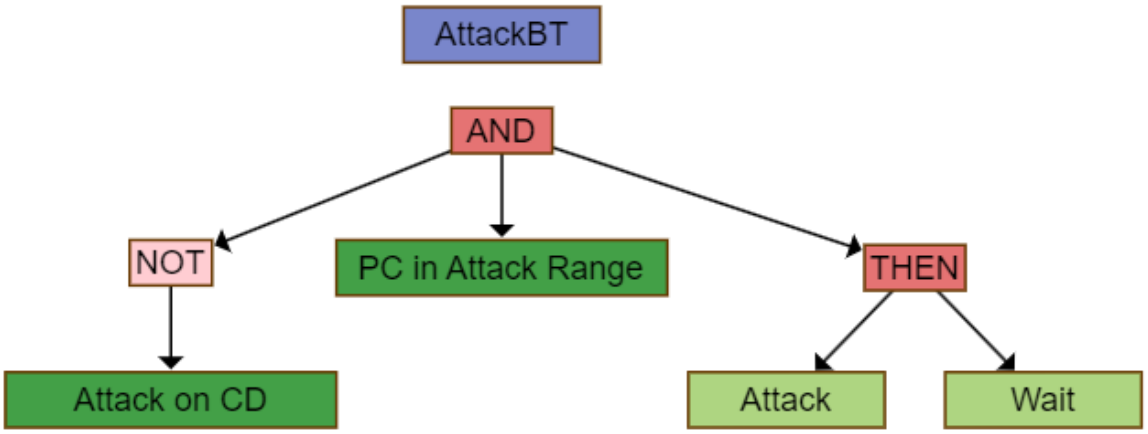

Figure 2: Attack subtree

The dash attack and dash subtrees are similar to the above.

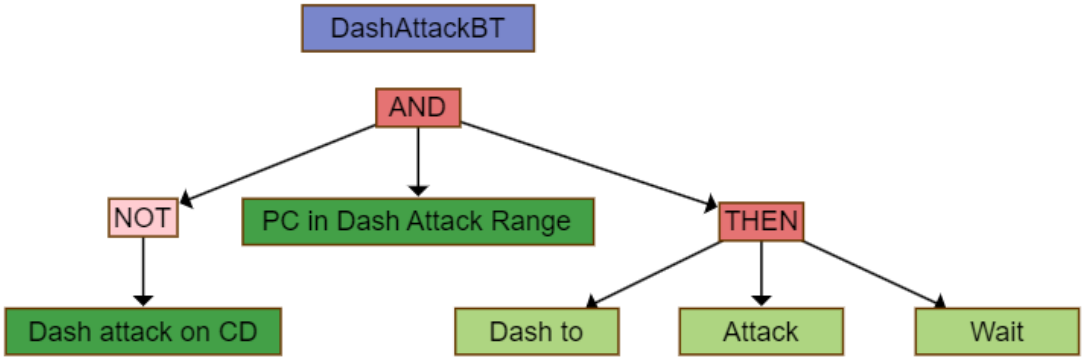

Figure 3: Dash attack subtree

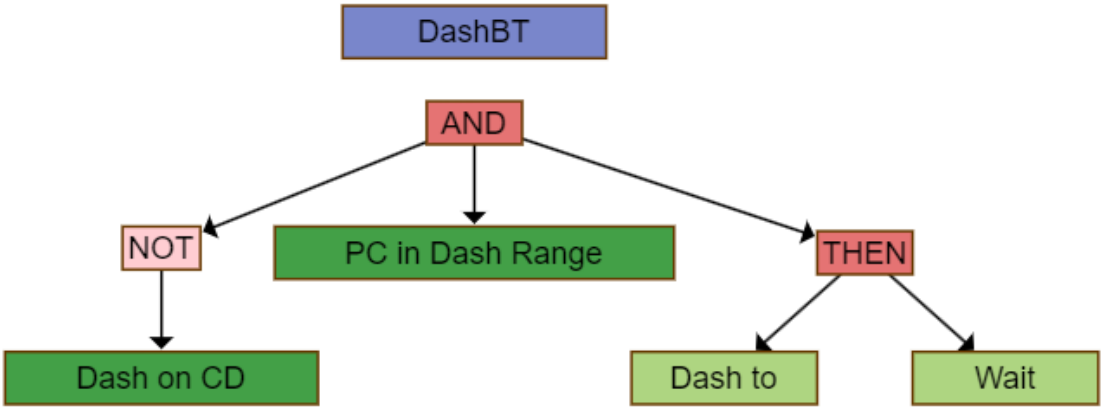

Figure 4: Dash subtree

Some examples of the final character trees are then the following:
The very first enemy, Nick, has patrolling as his default behaviour and when he finds the player character he tries to dash towards it and use his melee attack.

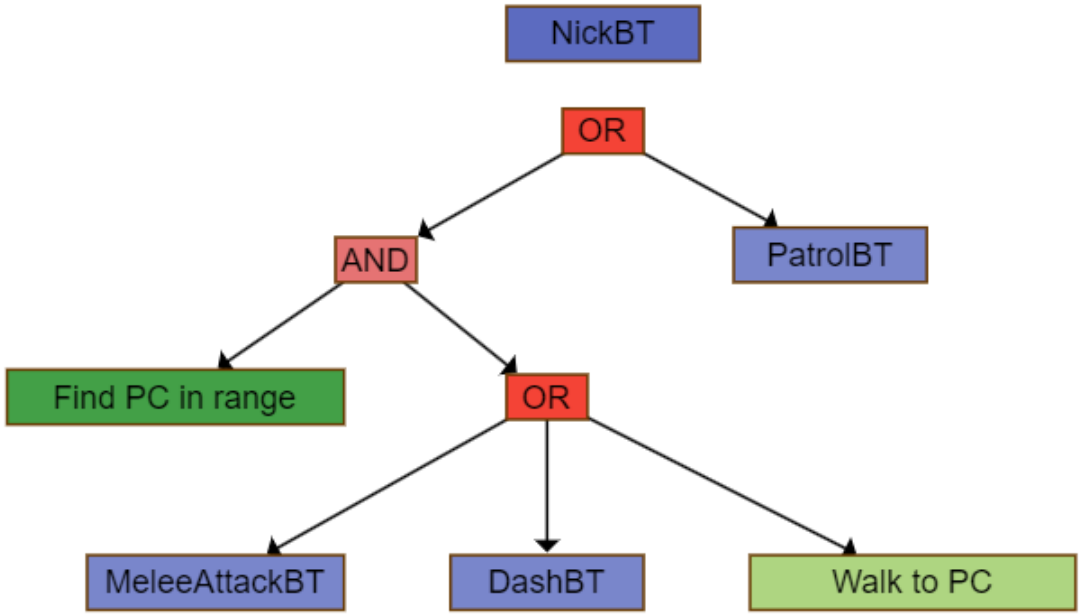

Figure 5: Nick tree

A slightly advanced enemy, Ms. Cups has two different attacks - one melee which pushes the target away (for when the player character gets too close) and one ranged.

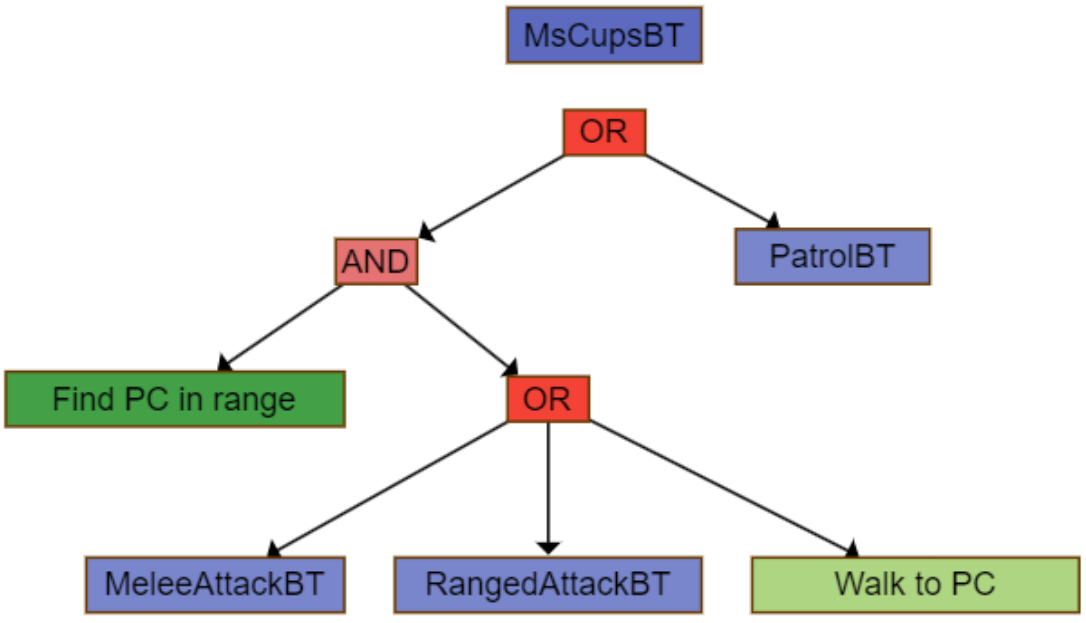

Figure 6: Ms. Pots tree

For pathfinding (i.e. the *WalkToTarget* task in the behaviour trees), the package A* Pathfinding Project (https://arongranberg.com/astar/) is used - for special rooms the navigation graph is pre-generated manually in the scene, for procedurally generated rooms from a *MapGenerator* the script *AIGenerator* generates the navigation graph according to the what tiles in the room's grid are empty.

Editor Scripting

Editor scripts are used mainly for setting up the graphics, with *GFXSetUpWindow* being the entry point - mainly sprite organization (*SpriteOrganizer*, *SpriteSetUp*) and animation generation (*AnimationClipGenerator, AnimatorGenerator*).

Procedural generation

Addendum by: Veronika Petrova

Since procedurally generated rooms are one of the pillar stones of the rogue-like genre we wanted to include them as well. Each level has a slightly different style of generation. It can be broken down to tile generation, extras like colliders and doors generation and finally object generation.
The scripts for each level inherit most of their features from base class MapGenerator.

The Office

The first level of the game - The Office - always consisted of two rooms, left and right. These were generated in a way, so the player couldn't get "behind the wall" - the corner always facing up.

The generation process for floor tiles could be summarized like this:

1. Pick two random numbers for each room - height and width.
2. Compare the two dimensions and decide what type of room it is
    a. room, horizontal hall, vertical hall
3. Now compare the two rooms together
    a. if the height difference is under a certain threshold and they are both "rooms", create one big room
    b. if they are the same type of hall, pick a middle height/width for them and combine them into one long hall
4. Decide on the start coordinates for the two halls so they
    a. don't overlap
    b. fit together nicely (avoid corners facing down and small overlap)
5. Pick a random tile for each room and fill it with the picked tile.

The overall process for generation of The Office level and the other levels loosely follows these steps:

1. Generate floor tiles

2. Generate walls
    a. where appropriate
3. Generate colliders
    a. surrounding the entire map
4. Generate doors
    a. entrance and exit
5. Generate objects

For this level, the objects were generated in two rounds - first were extra objects - pots and bins - lining the walls. This was used more to create an office feel than give any obstacles to the player. For the halls, no more objects were generated. However in rooms, there are also tables generated. Those were generated simply by how many could fit in the room, randomly picking the facing side from four combinations. The generated tables were then randomly given a chair and a specific table sprite.

Mill Street

In structure, the generation follows the same formula as for the first level. However, instead of the left room and right room, the rooms are separated into a main room and an extra room. The main room is always generated, but the extra room is only generated sometimes. The room types for this level were - street, plaza and newly crossroads.
Each time the map is generated it also rolls if it's a park or not - for a plaza this means that it's just a park or a plaza. For streets - which were separated into a sidewalk and road part - this decided if the road was an actual road or grass.

The room and hall combination is prohibited, but down facing corners were allowed. And also newly added crossroads, which were always made from just sidewalk tiles.

The street doesn't have walls, but instead is lined with buildings. These are placed only on walls that won't overlap the player and that make sense (they're not placed on the road part of the street for example).

The objects on this level are a bit more sparse, so the player has more space for combat and since streets aren't generally very crowded.

DLF Plaza

The last level is very similar to the second level in structure. With the exception of there being no rooms, only hallways.
It again doesn't have any walls, instead having storefronts that are evenly placed on each wall that makes sense (doesn't overlap the player).

The only generated objects on this level are the pillars. At the start, the generator randomly decides if there will be any pillars. If yes, it proceeds to randomly choose the gap between them related to hall width.

Special rooms

All of the special rooms - merchant and boss rooms - are handcrafted.