COM6013M – Artificial Intelligence for Games Assessment

Game introduction

Fortress Attack 2: Electric Boogaloo is a game created with the sole purpose of experimenting the influence of a game director that is constantly adapting to the player's state, being able to access not only player information, but also performance, and a procedural content generation system that is designed to build a challenging environment for the player. Both of those components are accompanied by smaller scale AI systems that control agents to create an adapting and fair experience based on player skill.

Game description

The game chosen for this project is a third person ship flying simulator that incorporates nimble flying with the fast shooting of enemy ships and the conquest of procedurally generated sky fortresses. The setting of the game is in an initially empty scene that very quickly generates the first medium level fortress in front of the player, whose attributes are entirely based on the player's statistics: health, number of kills and number of bullets left.

With the instantiation of the first fortress comes the spawning of the first enemy ship that will start following and shooting the player when within range. The player can fly around using the movement keys and aim the ship using the mouse, as well as shoot either semi-automatic by tapping the left mouse button or fully automatic by keeping the left mouse button pressed down. The game then becomes a dog fighting simulator where both the player and enemy have the goal of destroying each other. Upon destruction of the enemy, the player will be presented with a small amount of health and bullets, as well as a new enemy to fight. If the player is destroyed however, the scene restarts and a new game can begin.

Upon finishing off all of the enemies created by the fortress, initially 10, the fortress is marked as completed, destroyed and the player receives an amount of health as a reward, as well as a new fortress that now spawns based on the current statistics of the player, either becoming weaker if the player has low health and bullets, or harder if contrary. Also, the fortress respawning is virtually infinite and a new fortress can be generated either upon completion (destruction of all enemies) or upon leaving a set boundary away from it, in which case a new fortress will be spawned in front of the player.

AI system description

The main purpose of the procedurally generated fortress is to create an ever changing and unique object of interest, not only visually but also mechanically by offering greater challenges for those who have the skill to conquer the fortress, or an easier experience for those who are taking it lighter or are inexperienced. Being highly customizable due to the block-based procedural generation used, a fortress offers good scalability and efficiency at low costs, its generation being entirely controllable by alterable variables.

The player should both feel rewarded by defeating the enemy fighters and receiving health and ammo, as well as from needing to adapt to the evolving difficulty of the fortresses and the need to hone their skills when dog fighting other ships. For players with low performance, the opposite is available, an experience that is lighter, provides less enemies, less damage taken, less enemy health and generally more room to play and figure out the game's mechanics and tactics when engaging the enemy.

To be able to match the criteria of offering an optimal and unique experience for each player, the game director is used to heavily influence the generation of the fortress and the attributes of the enemies, being able to control the amount of damage dealt by them and their health. The game director adapts with every fortress generation and is able to keep track of the player's progress throughout the game, increasing or lowering the challenge level to create a rewarding experience rather than either boring the player with easy levels or infuriating them with impossible difficulty.

Finally, the enemies, whether dynamic (ships) or static (turrets) are the challenge that the player needs to defeat to get the satisfaction and fulfilling feeling of joy when playing the game, therefore their implementation has an important impact on the final player experience. Even if PCG and the Game Director are the main highlights of this project, the finite state machine system (FSM) used to program the enemy ships really helps bind the two symbiotic elements together by offering powerful controllability and moderate scalability, while still being entirely guidable by the game director's influence.

Implementation

Game development

The playable character of the game is a third person ship that has the ability to move around and shoot using keyboard and mouse inputs. For the movement, physics and transforms manipulation using vectors and forces were used, with the input taken from the "W, A, S, D" keys while rotation and shooting control were taken from the mouse. The scene management is simple, the game only having 2 scenes, a menu that takes the player to the main scene and the actual main scene where the gameplay happens.

AI systems development

The PCG system is using matrices and loops to generate a floor grid and for each block of the grid it can generate walls, towers, and barracks, with some extras available to the towers in the form of weapons and to the walls in the form of decorative items. Most of the procedural generation is randomized, tiles being placed based on their location, with walls only being placed on the square floor's edges, the floor tiles only being placed inside the fortress and the filler items like towers and barracks only spawning on certain tiles.

The game director uses methods of both gathering the information from the player and returning it to the procedural generation, as well as ways of providing information to the enemy agent to either enhance its attributes or its performance.

Finally, the FSM used is an efficient way of creating the enemy agent's behavior and for a small scale game like this this approach is useful and entirely capable of supporting possibly a multitude of other agents in future updates due to the lack of complexity that they require for creating a good player experience.

Procedural generator (PCG)

The PCG approach used in this project comes in the form of grid-based object instantiating and uses matrices, loops and randomization to create unique and controllable fortress-like structures in an efficient and procedural manner. While not using any evolutionary algorithms for content grading and evolution, the amount of variables that can be modify to change the layout and structure of a fortress makes this type of approach viable when going for the blocky aesthetic and always provides realistic and logically correct structures. Due to the nature of the player control and desired game outcomes, the direct interaction with the generated structure is mostly visual, as the actually generated tiles mostly contribute stylistically to the player and mathematically to the enemies.

Initially, the project was supposed to contain procedurally generated terrain, with the ability of having alterations generated by structures such as buildings, resources and enemy camps. However, this idea got completely overwritten when instead of only generating terrain and randomly placing preset buildings on top, it was decided to go for something more interesting that actually uses PCG to impact the game not only visually, but also mechanically. Therefore, the creation of a PCG system that puts together modular prefabs to not only create a fortress aesthetic but also give actual purpose to those generated elements was deemed a better idea for the better influence and control over the player experience.

```
1 reference
void CreateBase() //create the base (floor) of the fortress
{
    for (int i = 0; i <= maxRooms; i++) //run through the collumns of
    {
        for (int j = 0; j < maxRooms; j++) //run through the rows of
        {
            SpawnRoom(matricesPosition, i, j); //spawn a tile on each
            matricesPosition.z += basicRoomSize.z; //increase the z p
        }
        matricesPosition.z = roomSizeObject.transform.position.z;//re
        matricesPosition.x += basicRoomSize.x; // move to the next co
    }
}
```

Using the matrices system above, the creation of a scalable number of floor tiles, with the ability to add multiple shapes other than square in the future, became possible and allowed for the straight forward implementation of additional tile generation on the Y axis, on top of the floor tiles. The base system takes the *i* and *j* values as counters and coordinates of blocks on floor grid, while the use of another counter, usually still *i*, helps in the other height-based tile placing methods.

```
void CreateWalls(Vector3 wallsPosStart, int maxHeight, int i, int j) //method
{
    int randomVariation = Random.Range(0, maxRandomness); //again, get a rando
    Vector3 currentWallPosition; //the current position of the selected tile (
    currentWallPosition = wallsPosStart + new Vector3(0, basicRoomSize.y, 0);
    GameObject wall; //the wall tile to be created
    for (int k = 0; k <= maxHeight - randomVariation; k++) //loop that runs th
    {                                                      //max height minus
        wall = Instantiate((GameObject)floorsWalls[2], parentTransform);
        wall.transform.position = currentWallPosition; //create the wall and s
        if(Random.Range(0f,1f) <= decorationsPercentage) //take the chance of
        {
            CreateDecorations(currentWallPosition, i, j);
        }
        currentWallPosition.y += basicRoomSize.y; //increase the position to a
    }
}
```

The method above exemplifies the way the tiles placement works on the Y axis, using a loop to continuously add tiles, in this case wall blocks, on top of each other above the floor until they reach the height limit minus a random amount of randomness that offers some variation to the fortress' design. Moreover, all of the values can be tweaked to provide results in a wide array of creations, from being able to create tall and sharp walls by increasing both the maximum height and variation, to adding a multitude of assets that the fillers can be chosen from for maximized variation. For this project demo, there were mostly 3 prefabs to choose from when selecting a filler, accessory, floor tile, weapons and enemy plane types, but the system offers good scalability and can be easily tailored to any needs due to the efficient code used.

Game Director (GD)

The game director solution is used to help provide an adaptable and optimal experience for players of all skill levels, being able to manipulate the generation of the fortress directly and the enemy agent's attributes indirectly through the medium of the fortress generator. Classically, the game director influences the rewards and resources of players when they are progressing through a game, based on their performance and currently held resources, however the implementation of the game director in this project takes a different approach that aims to provide the same results for the player's experience.

Altering the fortress generation settings directly, the GD is directly interacting with the PCG system, harmoniously embedding all of the changes in the player's performance into the tile generation. By gaining more kills, the player is giving the game director the chance to increase the number of static turrets that will be attacking the player on the following generated fortress as well as increasing the chance that these weapons are added to the towers, therefore providing a bigger challenge for a more skilled player.

Attributes like health and bullets help the game director decide whether the next fortress will have more enemies and how much health those enemies have, as well as the total number of barrack tiles that will be spawned behind the closed walls. To further increase the difficulty, the number of randomly generated barracks, affected by the player's ammo, adds a small amount of health to every enemy that is spawned from that fortress, therefore a player who uses ammo efficiently will have to face tougher enemies which can possibly lead to more complicated strategies being developed by the individual to be able to survive and thrive.

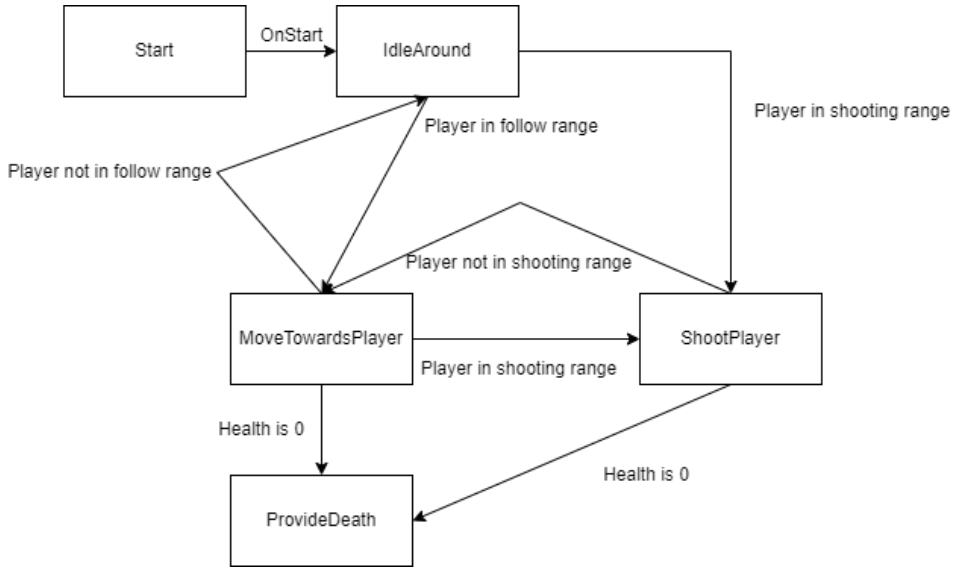
On the other hand, an inefficient and less skilled player with low health and ammo will be greeted by a notably smaller fortress, with less chances of barracks and towers being created and less enemies to fight, therefore easing the stress and increasing productivity. The constant dynamically changing difficulty is what can help an infuriating player regain their satisfaction by completing a smaller fortress and getting the health and ammo rewards from defeating fewer enemies, which can give them the skill and confidence necessary for taking on bigger challenges.

```
void AdjustFortressSettings() //get all of the settings of the fortr
{
    gameDirector.GetPlayerInfo(); //update the game director's infor
    maxRooms = (int)(gameDirector.playerHealth / 2.5f); //set the fo
    weaponsPercentage = (float)gameDirector.playerKills / 20;
    maxWeapons = gameDirector.playerKills;
    towerPercentage = (float)maxRooms / 300;
    barracksPercentage = (float)gameDirector.playerAmmo / 3000;
    decorationsPercentage = (float)gameDirector.playerHealth / 300;
    fortressHealth += gameDirector.playerHealth;
    fortressPlanes = gameDirector.playerHealth / 10;
}
```

In the picture above there are most of the variables that the player's stats affect directly, their formulas being obtained after thorough testing and evaluated to being as close to optimal as possible given the state of the demo. The method first calls the game director to get the latest player attributes and then assigns them to all of the meaningful factors that affect the fortress generation.

Finite State Machine (FSM)

While the PCG and the GD are the stars of the game, the FSM used to add behavior and dynamism to the enemies is very much impactful on the player's experience directly due to being the only AI system that takes the player's attributes from the GD and fortress and challenges the player to an aerial confrontation, having the ability to swiftly put an end to the game.



In the diagram above, the FSM is described using states and conditions, providing the backbone that the enemy agent relies on to defeat the player. The behavior starts with the idling state where nothing happens, the agent is stationary and waits for the player to be in range. Following the approach of the player's ship, the agent will engage in pursuit and, if in shooting range, attacks using an automatic turret with infinite ammo. During the fight, the agent will always try to point towards the player and shoot, therefore providing a side effect of turning and dodging attacks which is not written into the FSM. Finally, the agent can stop shooting and give chase, or cease chasing altogether provided the player leaves the specified ranges.

Symbiosis between the generator and game director

The tight connection between the procedural generator and the game director is exactly what makes the chosen approach unique and easily expandable. Normally PCG has a random factor to it by definition and whilst randomness was used in this project, most of the generating influence came from the game director's data. Successful content generators usually follow strict heuristics that help restrict the type of content they create and mitigate errors, to help achieve the game's goals and aesthetic. However, in this game the player is directly responsible, for the most part, for how the fortress generates and how many certain tiles it holds, adding the possibility of unexpected results occurring.

Technically speaking, the heuristics required to generate meaningful and playable content from tiles are created and modified in real time by the user, and they will be different for every player depending on their playstyle, skill and will to progress. With more possible alterations the game director can make in the future on the generator, the more the experience will be in the hands of the player, from looks and style up to difficulty and progression.

Discussion

Approach description

The PCG approach used to create the fortress building in this project is certainly not among the most efficient techniques that have been developed over the years in this branch of AI, but for this type of game and for the aim of the system it performed well. In terms of actual content generation, it provided a wide variety of fortresses when testing the parameters and the most natural design was chosen, with medium walls and tall towers, although more choices are always available.

Being able to modify a lot of the parameters that influence the fortress generation provides an impressive generative space, with the ability to easily modify some of the code even and create different types of shapes, with different rules. The heuristics used to determine what content the generator creates were more than sufficient for this game scale, but at the same time provides opportunities for future development and expansions.

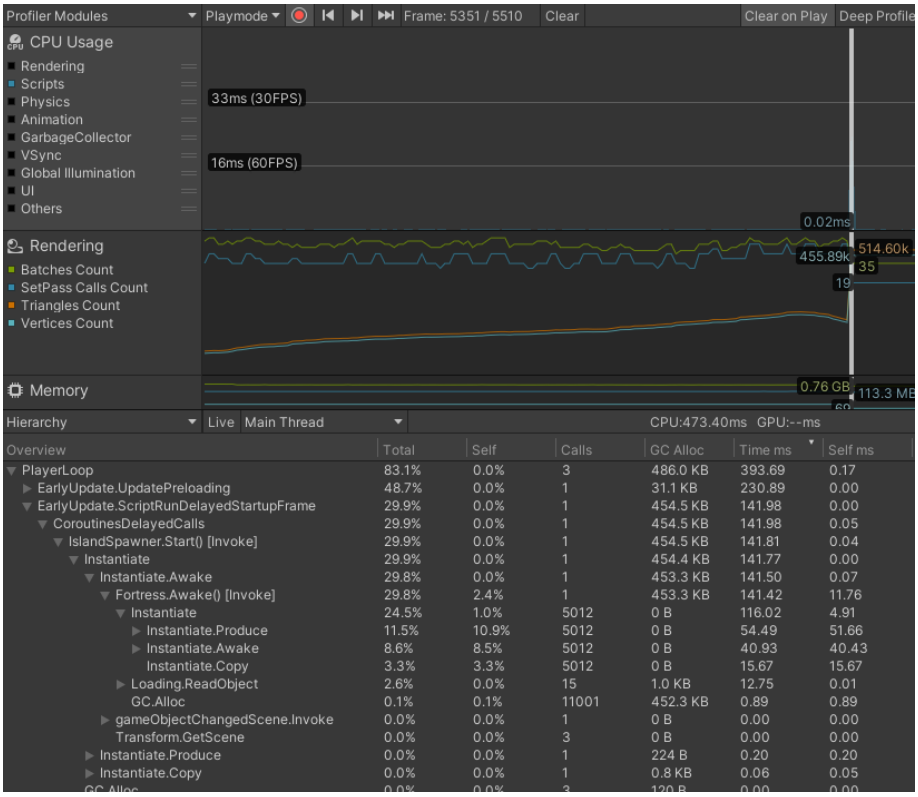
The GD, in tandem with the PCG system, not only provided a larger playable generative space with logical levels being built even under the influence of many variables, but also allowed for the player to tailor the game to their skills, and after testing the generator by simulating different skill levels, it has been noticed that the system handled everything regarding instantiation and adaptation as expected, therefore proving it's a viable solution when creating special adaptive experiences in video games.

The use of the FSM, while having little direct impact on the player itself, really helped weld all of the AI systems together by being efficient and minimizing the need for extensive game object interactions. As

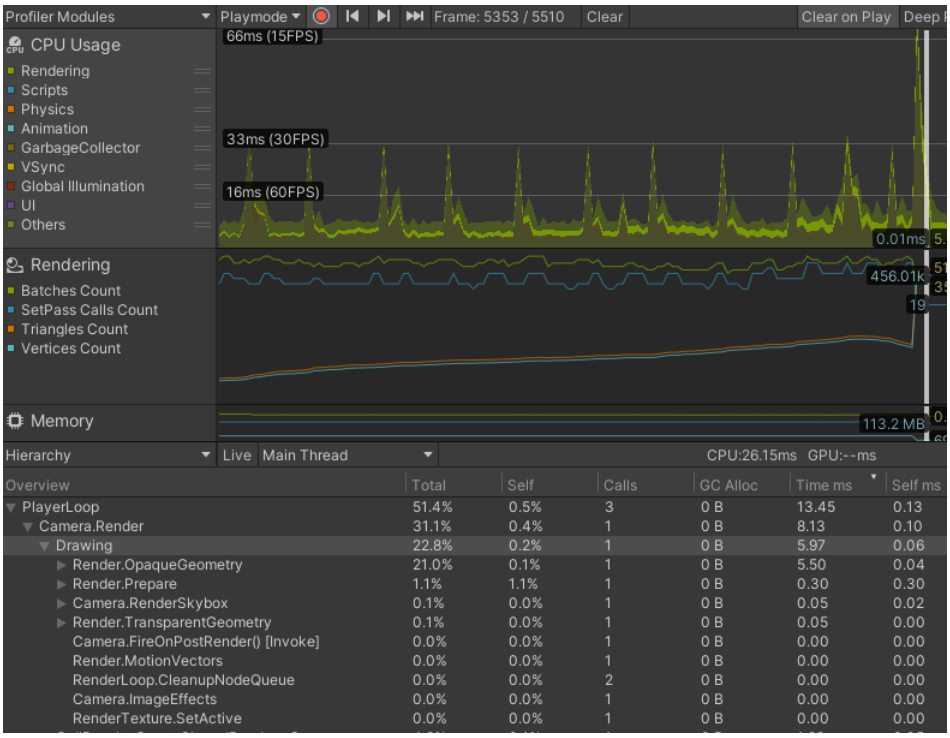
such, the use of a FSM proved to be a good idea, especially for the small size of the demo, still being able to provide accurate and controllable behaviors for the player to react to accordingly.

Approach evaluation

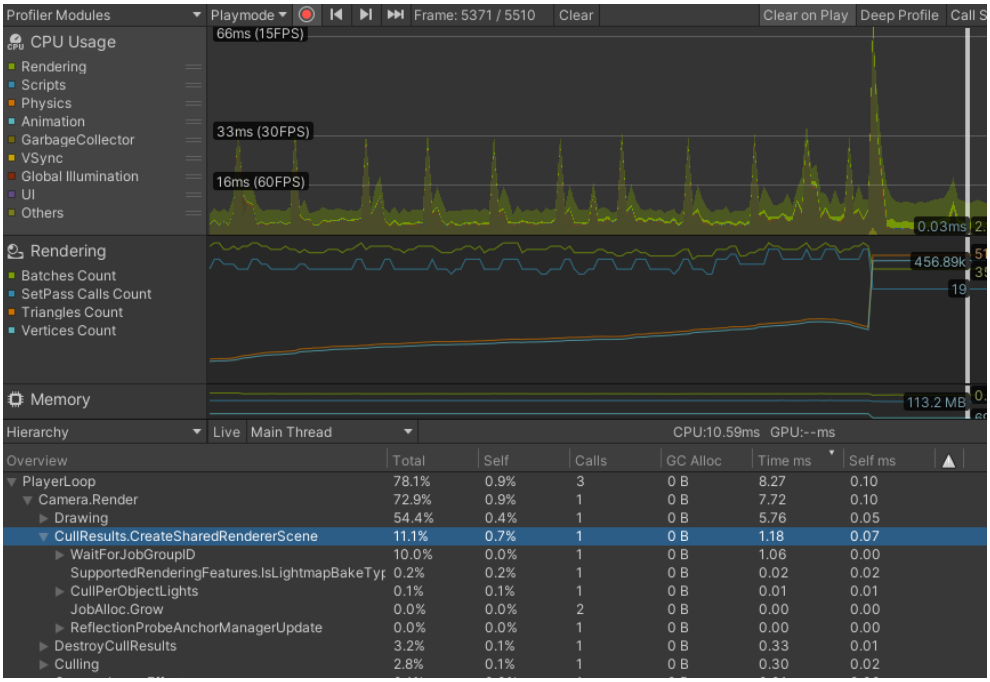
The technical and mathematical requirements of the PCG have made this approach somewhat demanding, at least in the generation phase when it calculates all of the tile positions and uses a multitude of loops while also instantiating objects, assigning materials and modifying variables. However, after using the Unity built-in profiler, running the game and performing some basic tasks like flying around, looking at the whole fortress, shooting the enemy and using collisions, it has been discovered that even if a lot of computational power goes to these calculations, after the fortress is generated there isn't much strain coming from the scripting side.



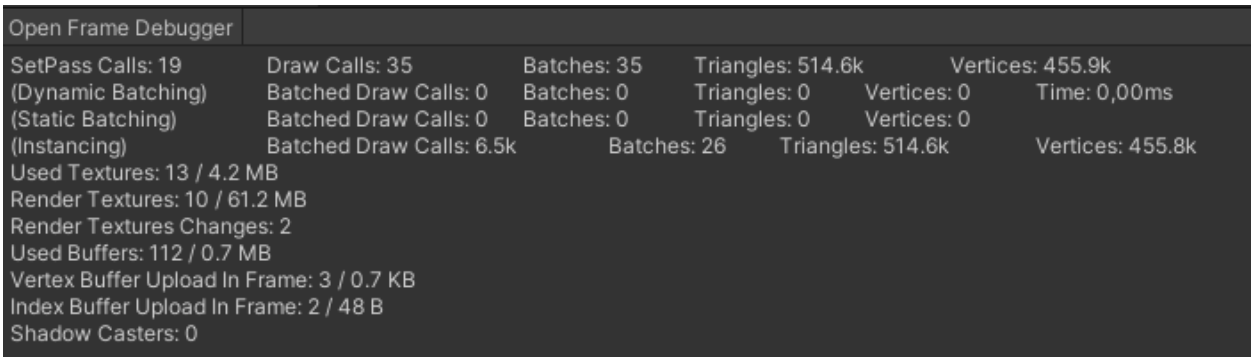
As observed above, most of the usage comes when creating the fortress due to the high volume of objects being instantiated, the base number being 400 floor tiles, 800 walls and around 1300 filler objects for the first medium fortress which obviously makes the computer work more, consume more resources and space in those frames, but after the generation most of the load is caused by rendering and culling the objects.



Above is the profiler view right after the fortress has been generated, and as it can be noticed before that, most of the processing was done for rendering and garbage collecting. The rendering was used intensely due to the large amount of objects rendered by the camera when looking at the fortress, while the garbage collecting was possibly intensely used when destroying bullets, running additional checks in code that didn't have to run necessarily, such as calculating the player and enemy distance when the enemy was already close enough to shoot the player.



One of Unity's built-in optimization techniques that really helped due to the nature of the fortress's design is object culling which disable the renderer of objects that are not in direct view of the camera, such as objects behind the player, or hidden behind other bigger objects. In the latter case, due to the modularity of the instantiated objects, the walls or towers could easily overshadow other objects and cause culling to reduce the usage of the renderer, providing optimal performance in that field.



Finally, as observed above, due to using GPU instantiated materials, the draw calls have severely dropped from a total of 6,500 to only 35, providing an incredibly noticeable performance boost compared to prior runs. Moreover, the framerate is constantly keeping up with updates and even when generating new structures, it does not fluctuate much, providing smooth gameplay for the player. A framerate counter has been added to the final build of the game to be able to monitor the performance in real time.

System limitations

Currently, the system provides good performance for the implementation used, however several optimizations are available not only for the PCG, but also for the GD. For the PCG, the costliest methods are the ones using multiple loops for generating tiles and constantly checking at every point, which can be optimized by grouping more loops together and instantiating more objects in the same loop, and possibly eliminating checks by adding separate scripts to specific tiles that should only have one type of filler on it.

On the topic of separating scripts, the generator lacks the modularity that separate classes can offer it and is currently holding all of the functions, making it inefficient when debugging or trying to modify a specific feature. The same can be said about the FSM, the lack of modularity can provide difficulties in the later stages of development, but it can currently be easily achieved.

Finally, the FSM would eventually run into scalability issues if expanded further and further, especially for a game that relies heavily on exploration and content variety, therefore behavior trees can be much more useful for the larger scale that this project could have in the future.

Future and alternative solutions

With the field of PCG being so vastly researched and developed, there are bound to be more and more implementations and optimizations that outperform any currently used techniques, and that can provide amazing benefits when looking to expand a solution for generating content like fortresses, buildings etc. While the solution of generating tiles based on a grid using modular assets is a good way of creating a wide variety of controlled results, approaches like reinforcement learning and noise-based generators have a lot of different uses and documentation that can possibly provide better results even in the field of modular buildings. Moreover, evolutionary algorithms are becoming so popular and viable for use in content generation, being able to collect data and correct their creations based on evaluation, that such a solution would benefit even further from being altered by a game director.

The GD used is currently implemented by gathering raw data of the player rather than properly analyzing gameplay, choices and actions which can sometimes misunderstand a player's real skill, even if it's generally accurate. Due to this, a GD that uses more in depth analysis to decide how to alter the game environment is much more useful for creating an optimal player experience and can even be powered by real AI techniques such as machine learning.

Conclusion

In conclusion, despite the limited time frame, the limited assets created and the relatively low number of parameters used, the PCG algorithm has met its scope of providing playable, scalable and adapting content based on the payer's skill, with the help of the GD and thorough testing, making it a good starting point when creating AI with the purpose of tailoring the game experience to the player's needs.

With a large amount of variety and generative space, the PCG system working in tandem with the GD can create a truly immersive and rewarding experience, especially if future versions will contain more optimizations, generating abilities and possibly even real AI techniques like machine learning. There are certainly limitations that inhibited the system from attaining its full potential, and there may be better implementations that meet the same scope, but generating tile-based modular fortresses with the help of a game director and creating enemies tailored to the player that use a moderately scalable FSM for behavior control have allowed the player's experience to be customizable, therefore meeting the aims of the game.

Finally, living in the age of technology and ever-evolving technology is allowing PCG solutions to become more and more varied, customizable and offer more accurate results. Implementations are always being researched, developers are constantly looking for more optimizations and this project, while already meeting the scope of improving the player experience, can definitely evolve to become smarter and more efficient.

References - Assets

1. Scrycoast - Cope! Skybox Pack: https://assetstore.unity.com/packages/2d/textures-materials/sky/cope-free-skybox-pack-22252
2. Unity Standard Assets: https://unity.com/
3. Ship controller – gamesplusjames: https://www.youtube.com/watch?v=J6QR4KzNeJU