Project evaluation

Advanced Games Design and Development- COMP 1288

Kaisei Fukaya

ID: 000979004-7

Table of Contents

1. Introduction

The aim of this document is to evaluate contributions to the project, discussing changes made throughout development. These are separated into specialism and non-specialism-based contributions.

In section 2 of this document, specialism-based contributions will be discussed covering the implementation of level generation, item generation and dynamic difficulty adjustment (DDA). In section 3 additional non-specialism-based contributions will be discussed, and in section 4 topics discussed throughout this document will be concluded.

2. Implementation of Specialism

This section of the report will cover aspects of the product that pertain to the specialist area, exploring how each of the three implementations have been developed in relation to final product. This chapter consists of section 2.1, which covers the level generation system, section 2.2, which covers Item generation and section 2.3 which covers dynamic difficulty adjustment.

2.1 Level Generation

The goal for level generation was to adapt Prim's algorithm using a modular tile set. This works by propagating through a grid of points, creating connections between those points until there are no-longer any points left to alter (Pittman, 2015). Enemies and traps would then be spawned similarly to the game Spelunky (Yu and Hull, 2008).

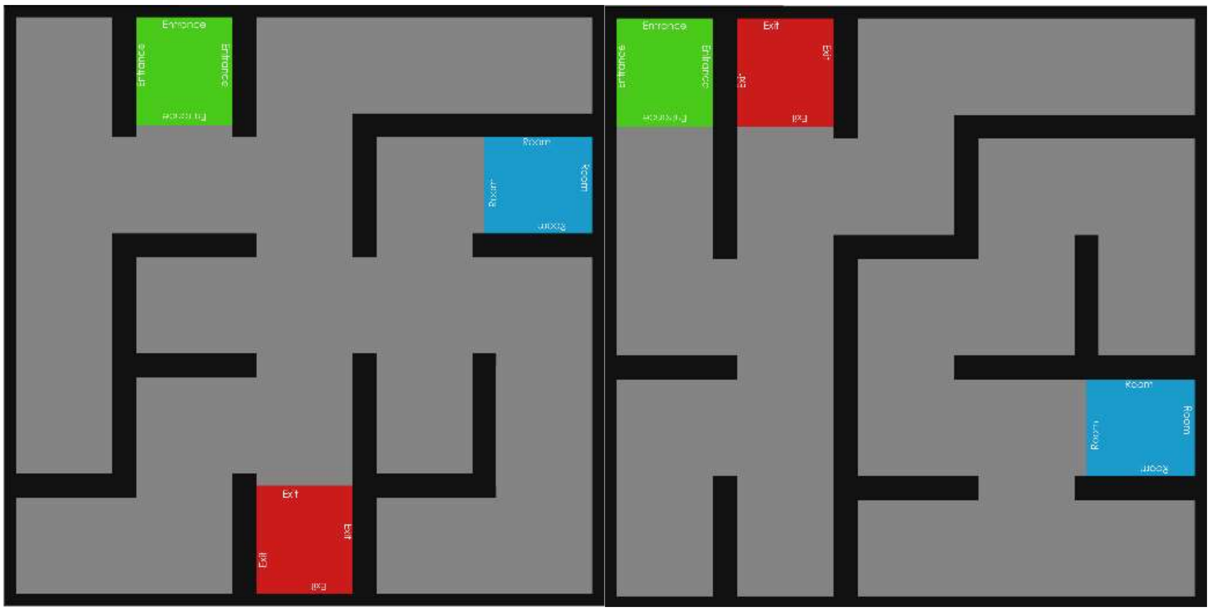


Figure 1: Early test of level generation using 2-dimensional grid. Good result (left), Bad result (right)

One adjustment that had to be made to the system early on was how many starting points the algorithm begins with. Initially, the entrance, exit and any rooms of importance behaved as start points. Each of which having the chance of propagating at any given step. This resulted in a high probability of rooms forming separate clusters where there would be no possible access to the rooms within- as shown on the right side of figure 1. This would frequently result in levels being broken as there was no possible route between the entrance and exit. Prim's algorithm, as discovered, only has a fully interconnected result if there is one start point.

To solve this issue the level is generated with the entrance room being the only start point. The exit and other important rooms are established and marked as occupied before any generation takes place, such that the algorithm propagates around these rooms. The result is a level where every room is accessible from the entrance- except for the pre-placed rooms. Granted that the algorithm has worked successfully it can be assumed that these rooms are not connected to any surrounding rooms. An additional step can then be added that iterates through each of these rooms and connects them to adjacent cells with respect to the side with a doorway. This results in levels more akin to the left side of figure 1.

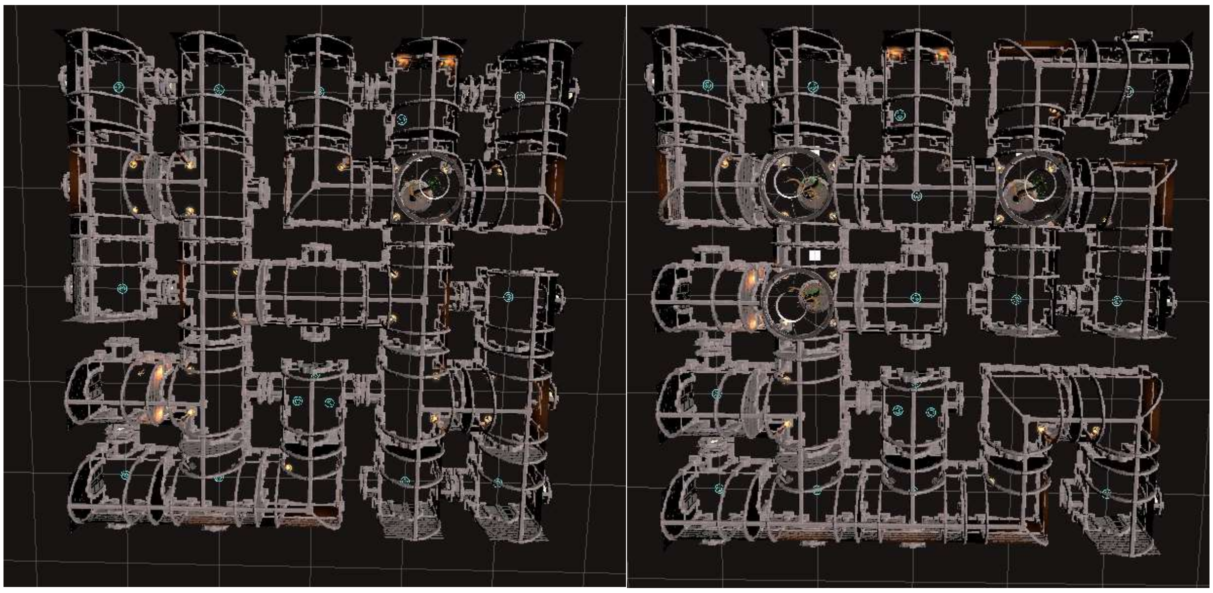
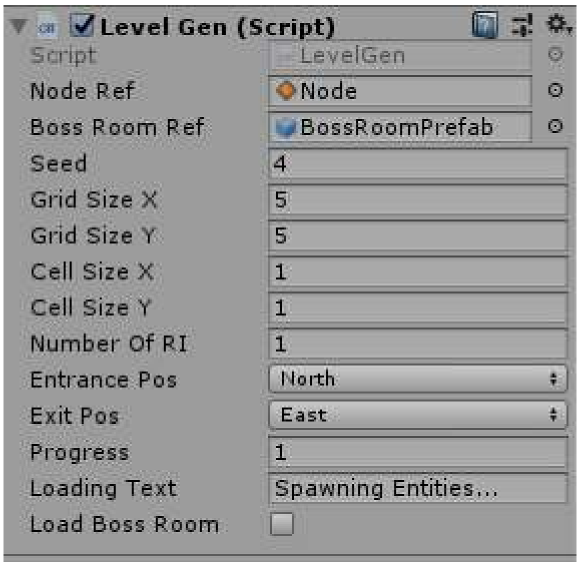


Figure 2: Two examples of final generated levels. All rooms are connected with an entrance (on the west edge) and an exit (on the north edge).

Figure 2 shows two examples of a levels generated using the final implementation of the algorithm. The algorithm first decides how each room is connected, then places the correct tile types at the correct orientations. This is done such that tile assets can be swapped out to produce levels using different content. Therefore, in the case of new content being added or changed the system will still behave the same.



Level Gen (Script)

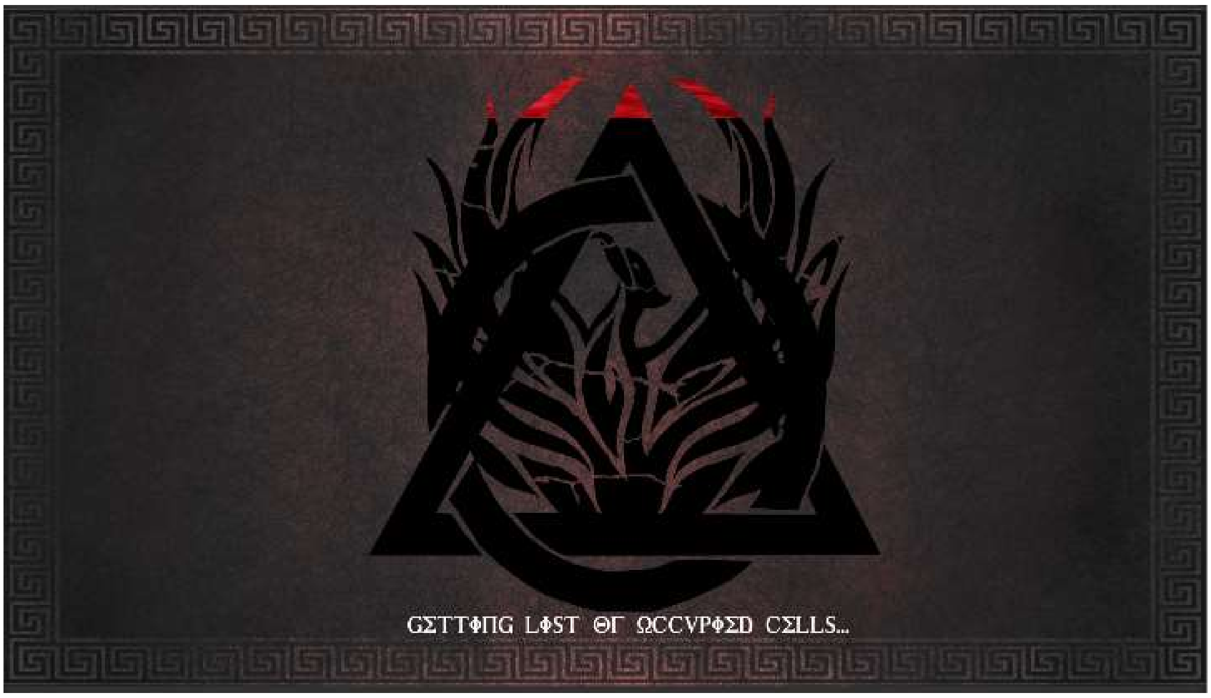
Script	LevelGen
Node Ref	Node
Boss Room Ref	BossRoomPrefab
Seed	4
Grid Size X	5
Grid Size Y	5
Cell Size X	1
Cell Size Y	1
Number Of RI	1
Entrance Pos	North
Exit Pos	East
Progress	1
Loading Text	Spawning Entities...
Load Boss Room	☐

Figure 3: Level Generator script within Unity inspector.

This is facilitated by access to the algorithm's properties, where values such as the seed- entrance and exit positions and number of important rooms can be edited. As well as access to node objects on which references to room prefabs can be changed.

During development, changes to the tile assets were made such that rotating a single room type would cause inconsistencies in the way rooms lined up. This was solved by having variants of rooms for each possible orientation to select from. This meant restructuring the way room prefabs are selected at the end of generation.

Finally, the entire algorithm was placed within a coroutine. A coroutine, as defined by Unity documentation, is a method that can suspend its execution (Unity Technologies, 2020). This means that code can be performed over the course of multiple frames, allowing other logic to occur at the same time. In this case the progress of the level generation can be presented to the user via a loading screen.



GΣTTΦΠG LΦST ΘΓ ΩCCVPΦΣD CΣLLS...

Figure 4: Loading screen during level generation, with text to show the algorithm's current state.

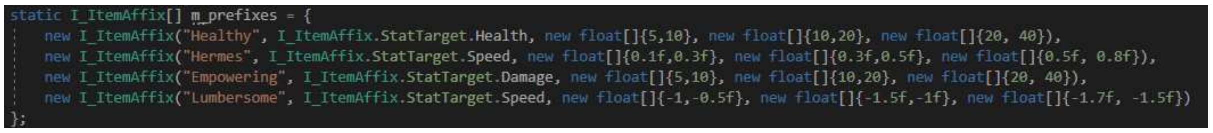
2.2 Item Generation

The chosen method of item generation is based on systems found in Diablo 2 (Blizzard North, 2000) and Path of Exile (Grinding Gear Games, 2013), where item stats are derived from a list of possible modifiers and base items.

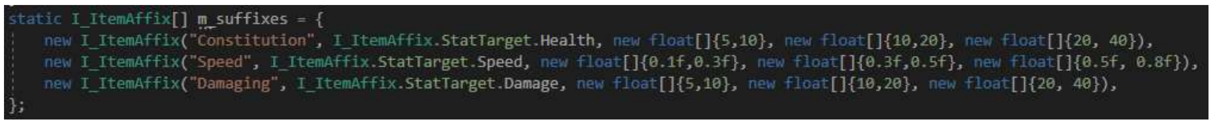
The final implementation results in items that consist of a base item and two modifiers, a prefix and a suffix. These form the name of the item while also indicating stat changes. An item is generated based on two key pieces information- the rarity tier and the item type. These are provided by the entity that is dropping said item. The rest is randomised and procedural.

First the base item is established, this is the core item that will later be modified. This is taken from a list depending on the chosen type, for example if the type is weapon- the base item could be a bow, sword or spear. These base items have baseline stats such that every item is consistent, with variation only being derived from modifiers.

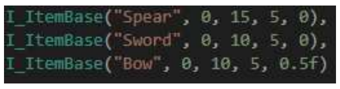
Modifiers or affixes are then added, these consist of a prefix and suffix. The prefix, forming the word or phrase appended to the base items name- is taken from a relevant list of prefixes. These consist of a name, a stat target and value ranges for each possible tier. The stat target value is used to express which stat should be modified, for example a damage-based affix will target the damage stat. The tier-based value ranges are assigned such that items of a higher tier will have higher stats in general with variability in the specific value assigned.



```
static I_ItemAffix[] m_prefixes = {
    new I_ItemAffix("Healthy", I_ItemAffix.StatTarget.Health, new float[]{5,10}, new float[]{10,20}, new float[]{20, 40}),
    new I_ItemAffix("Hermes", I_ItemAffix.StatTarget.Speed, new float[]{0.1f,0.3f}, new float[]{0.3f,0.5f}, new float[]{0.5f, 0.8f}),
    new I_ItemAffix("Empowering", I_ItemAffix.StatTarget.Damage, new float[]{5,10}, new float[]{10,20}, new float[]{20, 40}),
    new I_ItemAffix("Lumbersome", I_ItemAffix.StatTarget.Speed, new float[]{-1,-0.5f}, new float[]{-1.5f,-1f}, new float[]{-1.7f, -1.5f})
};
```



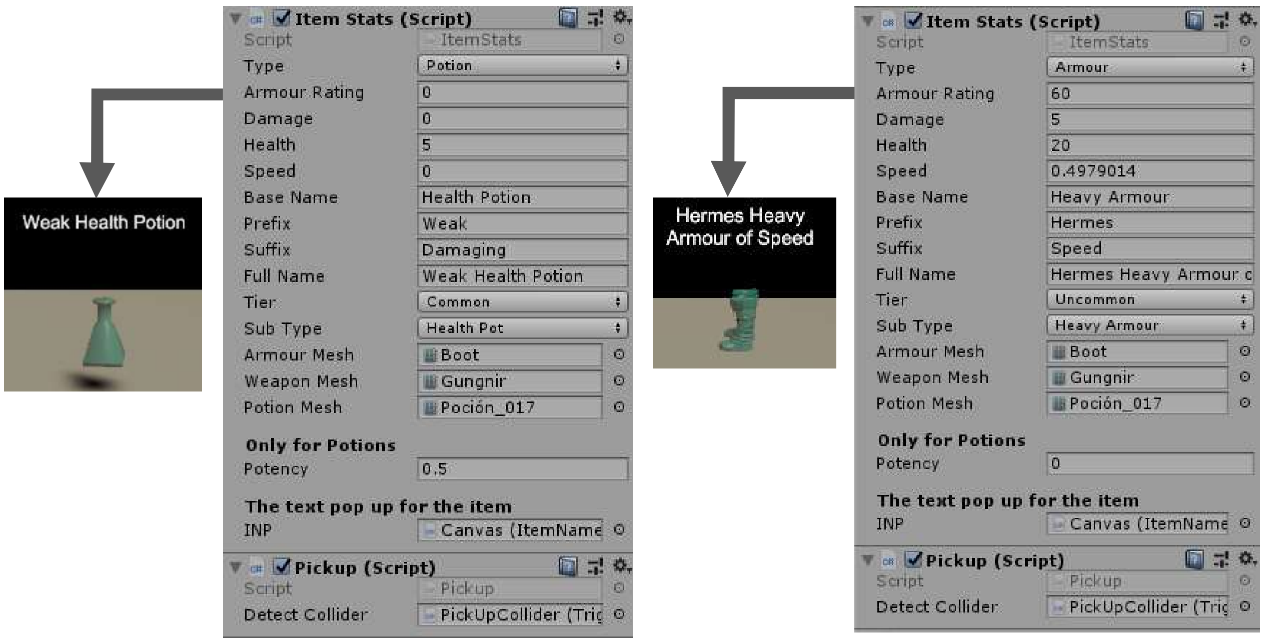
```
static I_ItemAffix[] m_suffixes = {
    new I_ItemAffix("Constitution", I_ItemAffix.StatTarget.Health, new float[]{5,10}, new float[]{10,20}, new float[]{20, 40}),
    new I_ItemAffix("Speed", I_ItemAffix.StatTarget.Speed, new float[]{0.1f,0.3f}, new float[]{0.3f,0.5f}, new float[]{0.5f, 0.8f}),
    new I_ItemAffix("Damaging", I_ItemAffix.StatTarget.Damage, new float[]{5,10}, new float[]{10,20}, new float[]{20, 40}),
};
```



```
I_ItemBase("Spear", 0, 15, 5, 0),
I_ItemBase("Sword", 0, 10, 5, 0),
I_ItemBase("Bow", 0, 10, 5, 0.5f)
```

Figure 5: Example lists of prefixes, suffixes, and item bases- in code.

The use of lists in this case allows for additional content to be added to the system without any modification to the code, scaling with a larger pool of content as well as allowing artists or designers to add content variations. Currently these lists exist directly in code, however this could in future be done through reading from files similarly to Diablo 2 (Blizzard North, 2000), and the existing system would not need to be altered to accommodate for this.



Item Stats (Script)

Script	ItemStats
Type	Potion
Armour Rating	0
Damage	0
Health	5
Speed	0
Base Name	Health Potion
Prefix	Weak
Suffix	Damaging
Full Name	Weak Health Potion
Tier	Common
Sub Type	Health Pot
Armour Mesh	Boot
Weapon Mesh	Gungnir
Potion Mesh	Poción_017
Only for Potions	
Potency	0.5
The text pop up for the item	
INP	Canvas (ItemName

Pickup (Script)

Script	Pickup
Detect Collider	PickUpCollider (Trig

Item Stats (Script)

Script	ItemStats
Type	Armour
Armour Rating	60
Damage	5
Health	20
Speed	0.4979014
Base Name	Heavy Armour
Prefix	Hermes
Suffix	Speed
Full Name	Hermes Heavy Armour c
Tier	Uncommon
Sub Type	Heavy Armour
Armour Mesh	Boot
Weapon Mesh	Gungnir
Potion Mesh	Poción_017
Only for Potions	
Potency	0
The text pop up for the item	
INP	Canvas (ItemName

Pickup (Script)

Script	Pickup
Detect Collider	PickUpCollider (Trig

Figure 6: Two examples of items generated.

Figure 6 shows two examples of items generated using this system. All generated values are shown on the right, with these items being represented in the game environment on the left. Generic meshes are used to depict dropped items as their variability is so large in comparison to the number of meshes that have been created for the game.

2.3 Dynamic Difficulty Adjustment

To improve the player’s sense of accomplishment, the difficulty of a game must be balanced such that it isn’t too easy or too hard. The game must accommodate for the variability in skills and knowledge between players, as well as changes to a player’s ability throughout the course of gameplay (Silva, Silva and Chaimowicz, 2015).

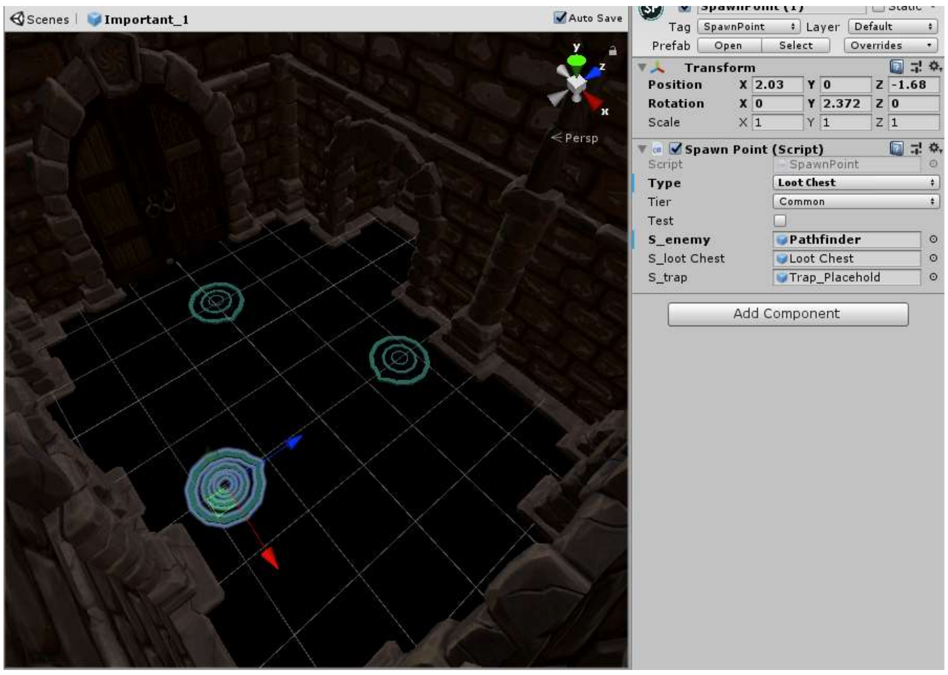
The dynamic difficulty adjustment system was not completed to the standard initially planned. The aim was to dynamically balance the difficulty of the game based on the ability and progression state of the player. This was to be done by interpreting information about the player character such as the quality of gear, how this gear is used and how far the player has progressed throughout the game. As this system relies on other systems to provide information, it cannot be fully implemented without such systems in place. Although the item and level generation systems have been designed with adjustment and dynamic game balance in mind, with properties being accessible such as item and enemy drop weights. No meaningful changes can be made throughout the course of gameplay without accurate, detailed and current information about the player character. In this case, throughout development other mechanics took precedence.

3. Additional Contributions

Additional features and tools outside of the given specialism were required to produce a functional product. This section will cover such contributions and how they have been developed. Section 3.1 will cover general tools, and section 3.2 will cover the audio system.

3.1 Development tools

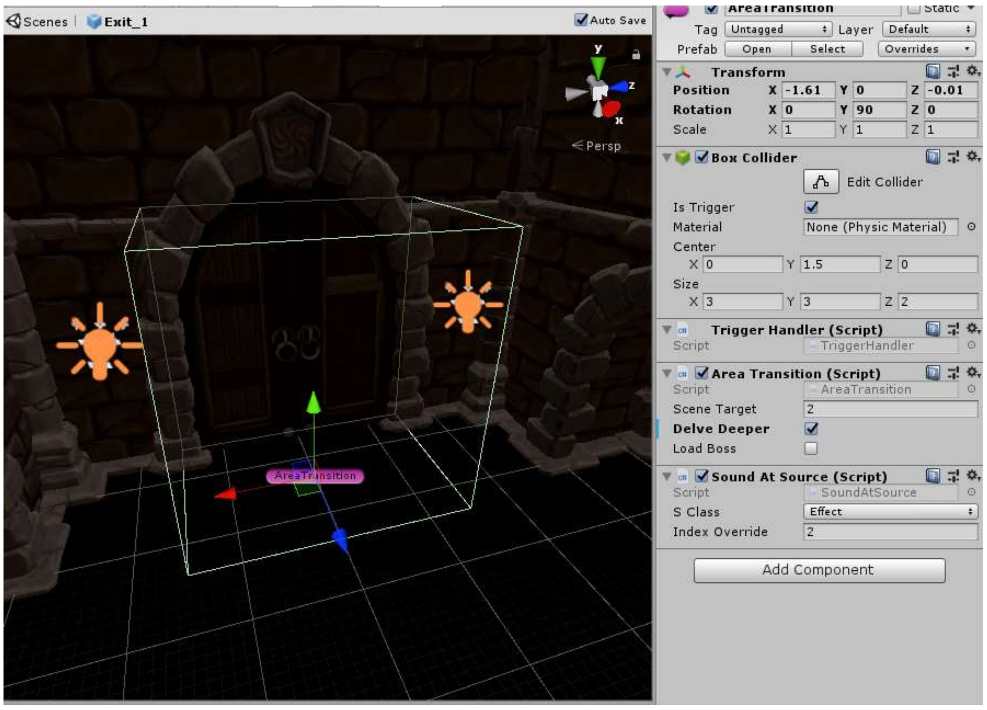
In order to help artists and designers implement level features consistently, some tools are provided with values that can be adjusted from the Unity editor without having to write code. One such tool is the spawn point, which can be set-up to spawn loot chests, enemies or traps. A simple 3D model is used to signify where the object will be spawned and what direction it will be facing. Additionally, the referenced objects can be changed so that different objects can be spawned- this is to allow for different enemy or loot variants to be easily selected. At the end of level generation, these spawn points are activated, such that their designated objects are instantiated in the exact position provided. This also ties in with the DDA and item systems as the system can choose how many of these spawn points are activated in a given level. For example, a specific number of enemies or loot chests could be designated based on the character's current state, so that less enemies and more loot chest are spawned when the player has weaker items. In this case only a subset of the pool of possible spawn points are activated.



```
Tag      SpawnPoint          Layer  Default
Transform
Position   X 2.03   Y 0       Z -1.68
Rotation   X 0      Y 2.372   Z 0
Scale      X 1      Y 1       Z 1
Spawn Point (Script)
Script        SpawnPoint
Type          Loot Chest
Tier          Common
Test          ☐
S_enemy       Pathfinder
S_loot Chest  Loot Chest
S_trap        Trap_Placehold
```

Figure 7: Example of spawn points within Unity scene.

The player spawn point works similarly, but instead attempts to spawn a player character only when no player character is found. If the player character is found, then it moves it to its position. This is done to facilitate the persistence of the player inventory between scenes.



AreaTransition Static
Tag Untagged Layer Default
Prefab Open Select Overrides

Transform
Position X -1.61 Y 0 Z -0.01
Rotation X 0 Y 90 Z 0
Scale X 1 Y 1 Z 1

Box Collider
Edit Collider
Is Trigger ✓
Material None (Physic Material)
Center X 0 Y 1.5 Z 0
Size X 3 Y 3 Z 2

Trigger Handler (Script)
Script TriggerHandler

Area Transition (Script)
Script AreaTransition
Scene Target 2
Delve Deeper ✓
Load Boss ☐

Sound At Source (Script)
Script SoundAtSource
S Class Effect
Index Override 2

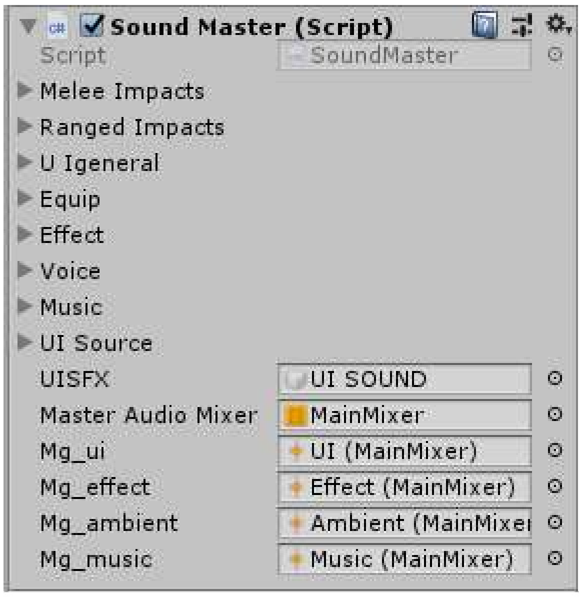
Add Component

Figure 8: Example of area transition object within Unity scene.

Another tool that was implemented are area transitions. These are objects that when the player comes near can be activated to change scenes or trigger a new level to be generated. This works by incrementing the level generator's seed upon each use so that each level is unique.

3.2 Audio system

A sound system was required to trigger and manage sounds within the game. This works by having a central master script that stores and categorises all sound assets, and a separate script that can be



Sound Master (Script)
Script SoundMaster
Melee Impacts
Ranged Impacts
U Igeneral
Equip
Effect
Voice
Music
UI Source
UISFX UI SOUND
Master Audio Mixer MainMixer
Mg_ui UI (MainMixer)
Mg_effect Effect (MainMixer)
Mg_ambient Ambient (MainMixe
Mg_music Music (MainMixer)

Figure 9: The sound system as seen in the Unity Editor.

added to any object that needs to interface with said script. The master script also routes audio through different mixer channels based on the category of sound. This is done so that the player or designer can adjust the volumes of the channels in relation to one another to balance out the audio, rather than manually assigning volume to individual audio clips.

Sounds can be played at the source of a parent object or through the UI. This means that sounds that are positional, such as ambient flame sounds or combat impacts can be played at the source object such that the audio listener can perceive them dynamically based on where they occur. Alternatively sounds can be played through the user-interface (UI), allowing for sound effects to be remotely triggered by objects- such that if the object is removed before the sound is complete, the sound can still continue to play. The UI sound object contains multiple sound sources that are selected from upon trigger. This allows for multiple sound effects to be played at once via the UI.

4. Conclusion

To conclude, two of the three stated specialism contributions have been completed to a level that was set out in the independent learning plan (ILP). Although changes were made throughout development, the level and item generation perform as initially stated, with properties accessible to other systems. An appropriate DDA system could not be delivered on due to limitations on player analytics, although consideration has been made for this within the level and item generation systems. Additionally, toward the end of development responsibilities and priorities had to be rearranged to create a functional product. This meant that the DDA system, which is reliant on all other systems, had to be de-prioritised.

References

Blizzard North, 2000. *Diablo 2*.

Grinding Gear Games, 2013. *Path of Exile*.

Pittman, D., 2015. *Level Design in a Day: Procedural Level Design in Eldritch*. [online] Game Developers Conference. Available at: <https://www.gdcvault.com/play/1022111/Level-Design-in-a-Day> [Accessed 7 Nov. 2019].

Silva, M.P., Silva, V.D.N. and Chaimowicz, L., 2015. Dynamic difficulty adjustment through an adaptive AI. *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, pp.173–182.

Unity Technologies, 2020. *Coroutine*. [online] Available at: <https://docs.unity3d.com/ScriptReference/Coroutine.html> [Accessed 12 May 2020].

Yu, D. and Hull, A., 2008. *Spelunky*.