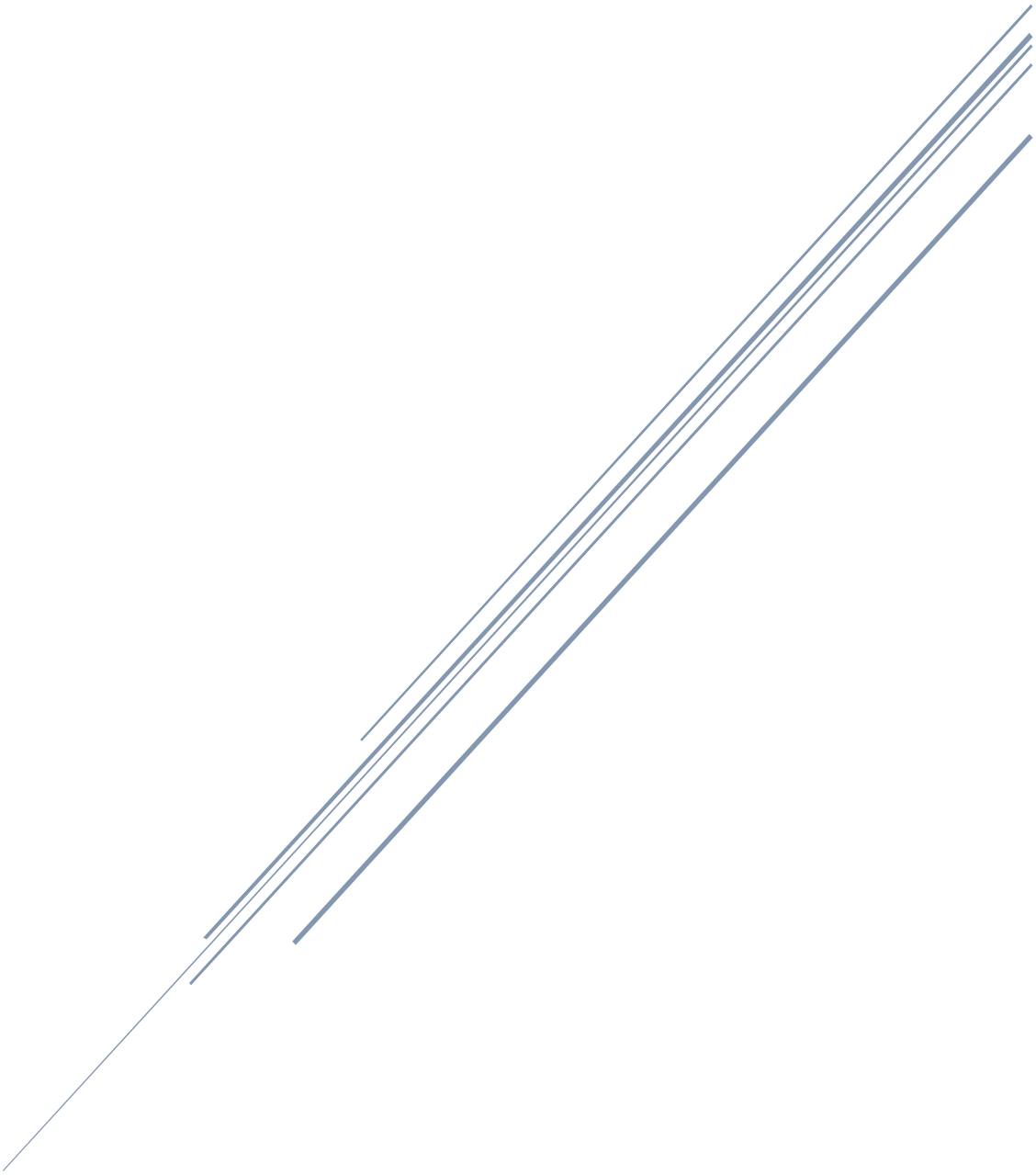
Name: Elzbieta Stasiak
Centre Name: Coloma Convent Girl's School
Candidate Number: 5053
Centre Number: 14310

CREATING A MENTAL MATHS PHONE APP

Computer Science Coursework



By Elzbieta Stasiak
Computer Science A-Level 2018

Contents

Analysis

Problem Identification

Currently many students struggle with mental maths; this is usually not a problem today when your phone can be used as a pocket calculator. However, in exams you are not allowed to use your phone and in some exams, maths in particular, a calculator is not allowed and students are expected to perform mental calculations under timed conditions. Students who struggle with mental maths take more time to do calculations in their head which wastes the limited time you have in an exam. If these students had a way of improving their mental maths that was fun and enjoyable, they would perform much better in non-calculator exams. For my project, I want to help students improve their mental maths through the use of games. I want to create a game because research has shown that while playing games you experience benefits such as: increased motivation and enhanced problem-solving skills according to the well-known psychologist Jerome Bruner (http://www.edudemic.com/game-based-learning-help-learn/). I intend to solve this problem by creating a game for Android phones that helps students to practice their mental maths and track their progress.

This problem is solvable using a computational approach because a computer can automatically check calculations to see if they are correct much quicker than doing questions from a workbook and having to check the answers at the back of the book yourself to check if they are correct. A computer can generate many different maths questions, so you never run out of questions to practice. It can also take into account what sort of questions the user wants to try, for example a user might want try questions that use addition and multiplication but not subtraction or division and they may want to use only two-digit numbers. A computer can keep track of progress and display it in a meaningful way such as using graphs and automatically update your progress.

Being a phone App is also an advantage because most secondary school students own a smart phone and can download Apps instantly and try the App, rather than buying a workbook and waiting for it to be delivered. Phones are also very portable, so a student could use the App during their journey to and from school without needing a table, paper and pencils to work with.

Identifying suitable stakeholders

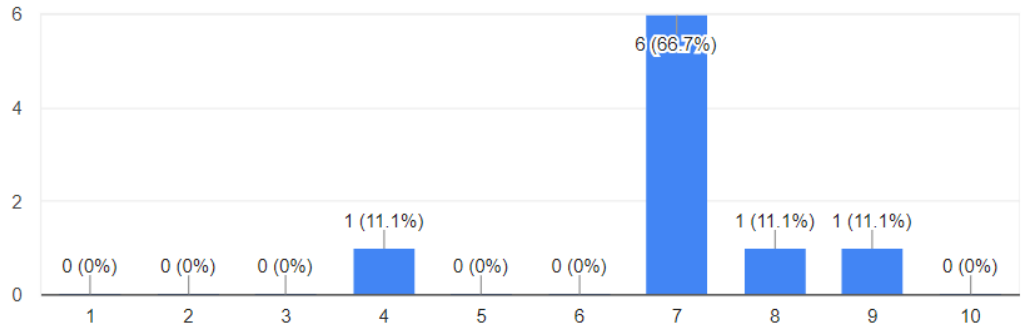
My stakeholders will be students in secondary school who struggle with mental maths and want to improve for exams such as GCSEs and A-Levels. My A-Level maths class of 8 people will be representing my stakeholders and I will use them as my user group to test my game because they will find it useful to use to practice their mental maths and improve it before taking their A-level maths exams in May and June. From discussing mental maths problems with my classmates, I have found out that they can do addition and subtraction of 1 and 2-digit sums but often need to use paper to help with bigger numbers and multiplying and dividing them. My game can help them because it will be a fun and interactive way to practice mental maths. It will also help them to be more confident for their exams and help them to quickly do mental maths in their exam when a calculator is not allowed.

I created a survey for my classmates to complete using google forms to ask them about their mental maths skills shown in the appendix on **page 138 & 139** the results of my survey are shown below (**Figure 1, Page 5**).

[Figure 1]

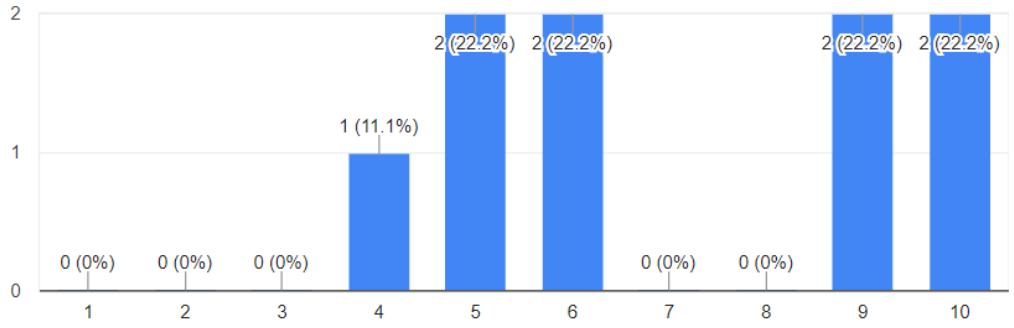
How would you rate your mental maths skills from 1 - 10?

9 responses



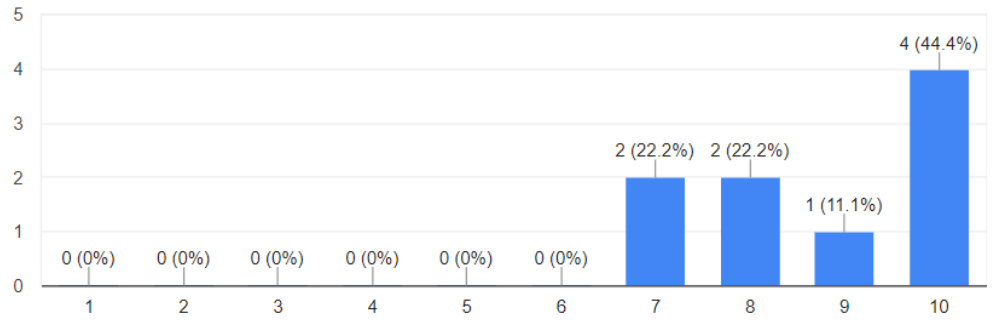
How important do you find mental maths in your daily life from 1 - 10?

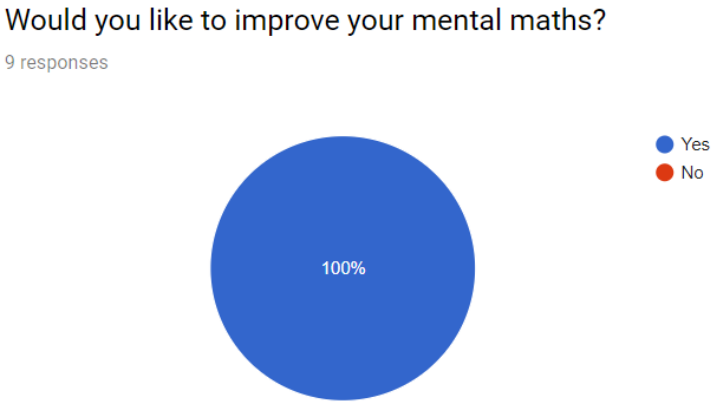
9 responses

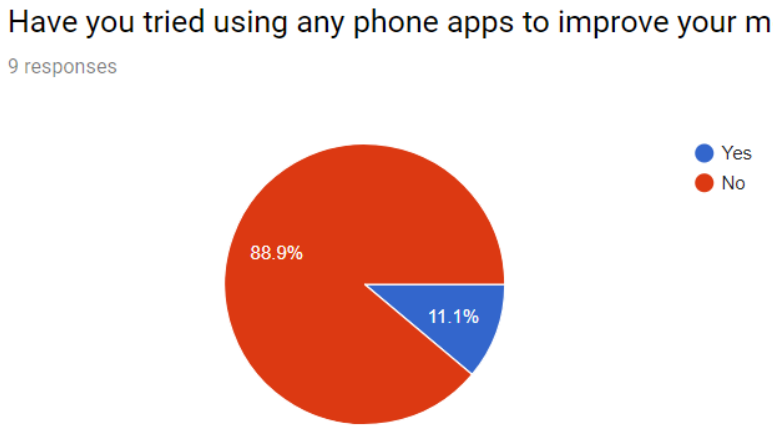


How important do you find mental maths for maths exams from 1 - 10?

9 responses





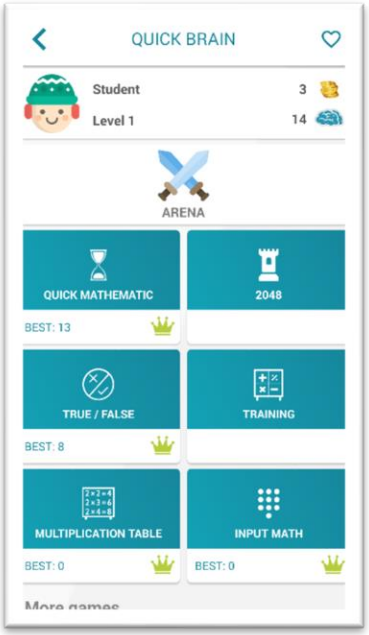


The results I got showed that on average people rated their skill level to be 6.7/10. They thought that mental maths was quite important in their daily life with an average rating of 7.2/10 but found it more important for maths exams as it received an average rating of 9/10. Every person wanted to improve their mental maths but only one person had tried using a phone app to do it. The person who had used a phone app said that they liked how the sums started small but increased in difficulty which helped them to increase their speed. There were no responses for the last question.

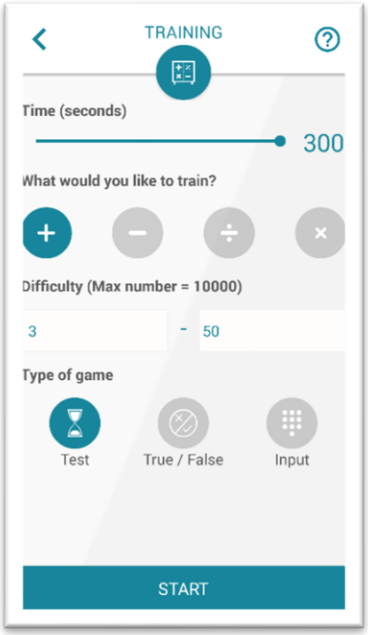
Existing solution research

I researched two Apps on the Google Play store that were mental maths oriented and two which were fantasy dungeon RPGs (Role Playing Game). The first mental maths game I looked at was called 'Quick Brain Mathematics – Exercises for the brain'. It is an App meant to improve your mental maths through several different exercises such as timed questions, multiplication tables and problems (**Figure 2, page 7**) where you must check if the maths is correct. Some features of the brain training app that I thought would be useful for my game, were the ability to choose mathematical operations (**Figure 3, page 7**) and track your progress to see if you have improved and keep track of best performances. I would like to include these features in my app because letting the user choose the mathematical operations and the range and size of numbers they want to practice with makes sure that the user can choose a level of difficulty that suits them, so they get a more individual experience because no one learns at the same pace. Also, being able to track their progress lets the user check if they are improving and lets them know what parts of their mental maths ability they need to work on more. However, I don't want to include a time limit because (**Figure 4, page 7**) I think the users would find it too stressful, and if they found the app stressful then they wouldn't want to play it and they wouldn't improve their mental maths.

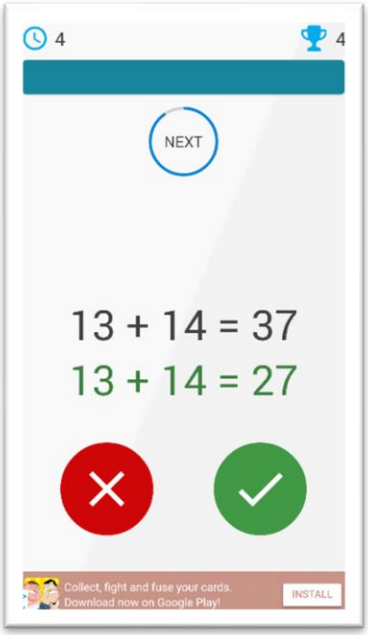
[Figure 2]



[Figure 3]



[Figure 4]



The second mental maths app I looked at was called 'Calculator the Game'. It is a mental maths puzzle game where the user has a couple of mathematical functions available to them and they must use them to calculate the target number in a specific number of operations (**Figure 5, page 8**) otherwise they can't progress to the next level. If they fail a calculation they have to keep trying until they solve it. I think this reduces its effectiveness as an App used to practice mental maths as you could just keep trying any number of combinations of operations until you get the target number (**Figure 7, page 8**), while just relying on your phone to do the calculations for you. Each level increases in difficulty with more complex operations and the solutions being less and less obvious. I like that this app doesn't feel like you are doing mental maths and there is no time limit which I found stressful in the other app I tested. It also had a clear, intuitive interface. From this app I want to include some of their approaches in my game such as the difficulty increase because if the difficulty stayed the same the user would never improve their skills. I also like the way they set up questions by giving you a target number and letting you figure out how to reach that number using mathematical operations and the specific number of moves you are given (**Figure 6, page 8**), rather than just giving you two numbers, you need to perform a mathematical operation on. I think this is a more fun way of doing mental maths, so I want to adapt this system for my game however, I will also limit the number of tries a person can use to try and get the target number to prevent the user from trying every combination of buttons and make sure they try and calculate the answer themselves before trying it.

[Figure 5]

[Figure 6]

[Figure 7]

The first dungeon crawler RPG I looked at was called 'Pixel Dungeon ML'. It is a simple game where you explore a dungeon and try to survive while collecting treasure (**Figure 10, page 9**). At the start you pick from one of four classes and you explore new areas of the dungeon by tapping where you want to go (**Figure 9, page 9**). You defeat creatures by tapping them to attack them (**Figure 8, page 9**) and they will chase and attack you back until you defeat or lose them. If you die, it is game over and you must start the game again. The game uses pixel graphics to display your character as well as enemies and a map layout of the dungeon, you can only view areas you have already explored otherwise they will be blacked out. I want to adapt the movement system in this game as I think it works well with touch controls because you can't use a keyboard with arrow keys with a phone. I don't want to use the same system for battling monsters however because it doesn't give the player much control over battles. I also want to adapt the graphical user interface (GUI) this app uses such as the dungeon layout display and health bar shown at the top of the screen.

[Figure 8]

[Figure 9]

[Figure 10]

The second dungeon crawler RPG I looked at was called 'Buriedbornes'. When the game starts you choose a class such as warrior or mage as well as an item to bring with you such as a healing potion (**Figure 11, page 10**). There are many different classes, most of which are unlocked through playing the game That give you different skills and equipment. You progress through each floor of the dungeon choosing between two paths each time (**Figure 13, page 10**). There are monsters in most rooms which you defeat in turn-based battles (**Figure 12, page 10**). Travelling through the dungeon you can gain better armour, weapons and skills (**Figures 14 & 15, page 10**) and defeating a boss allows you to progress to the next floor. I liked this style of dungeon better than the other RPG game because I liked the variety in creatures you come across and that you have more control over battles because they are turn-based. I think I can better adapt the turn-based battle system for mental maths and I will also adapt their system that procedurally generates dungeon rooms so that the dungeon is always different and never gets boring. The user interface of this app was the worst of the apps I researched and not very intuitive and it took a while to get used to, this could be because it was Japanese game that had been translated from Japanese (some of the writing was still not translated) and the text size was very small (**Figure 16, page 10**).

[Figure 11]

[Figure 12]

[Figure 13]

[Figure 14]

[Figure 15]

[Figure 16]

Essential features of my solution required

Summary of the features of my solution

I have decided to make a game that is a fantasy dungeon crawler that improves your mental maths. It will be a hybrid game that has the player exploring and trying to survive a procedurally generated dungeon using their mental maths. Monsters will try to attack the player and the player will try to defeat them using their mental maths skills. The target number will be the enemy's health and the player will try to reach that number in a specific set of moves using the mathematical operations given to them, otherwise the enemy will attack, and the player will lose a life. When the player loses all their lives it will be game over and the user will have to start a new game. Health potions can be gained from defeating enemies to increase the number of lives the player has. The player should also be able to view their high scores such as the number of enemies they have defeated and the most levels they have survived in the dungeon. This will give players a goal of trying to increase their high scores which will motivate them to keep playing and improve their mental maths.

User interface goals

To make this game I will need a Graphical User Interface (GUI) that allows the user to interact with my game which must work with touch controls since I am creating my game for Android phones. The user will tap the screen for movement, so I will need to create an algorithm that reads where the user tapped on their phone screen and uses pathfinding to move the player character to the corresponding area in the dungeon. My game will use 2D pixel graphics because they are simple to create.

Dungeon level goals

I will also require an algorithm to procedurally generate the dungeons in my game to make sure that each time a user starts a new game they have a new dungeon to explore with different enemies placed in different areas, so the game will not be repetitive and boring.

Generating maths questions goals

I need an algorithm that generates the mental maths questions and can generate different questions depending on the settings the user has picked. This is so that the user is continually challenged by not always doing the same questions. This is an advantage of using a computer because the user will almost never come across the same questions, so they will be able to use my App without running out of questions to try so they can carry on using my App for as long as they want to practice their mental maths.

High scores system goals

I will need an algorithm that keeps track of the user's progress. It will store the user's high scores in the game such as the least time taken to solve a maths problem and quickest time taken to complete one level of the dungeon or the whole dungeon. It also needs to keep checking for when the user beats previous records, so it can update the high scores.

Management system goals

I also need to create a system that manages the main game including all the algorithms it uses as well as the inputs and outputs of my App. It needs to manage and organise when each algorithm needs to be used and deal with the inputs and outputs of each individual algorithm such as making sure the algorithm that generates maths problems, generates them in line with the options the user chose and outputs the maths question to the different parts of the GUI to display the problem to the

user as well as display the operations generated as buttons. It also needs to monitor the user's touch inputs and update the player character's position to move it and display it to the user on their phone screen through the GUI.

Computational methods used to help me create my solution

- **Problem Recognition –** When programming the procedurally generated dungeons, I will need to use problem recognition at each stage of making the algorithm to make sure each separate method works correctly. For example, I will need a map of the dungeon to spot errors in the algorithm and be able to understand the variables that affect the map such as the positions and sizes of the rooms in the dungeon.

- **Problem Decomposition –** My solution will be split into many smaller problems that will have their own classes and each class will contain many methods such as when generating maths problems for the user, my algorithm will need to generate a target number, a starting number, the number of moves the player has and all the different operations the player can use. I can program each part separately and combine to form the complete program.

- **Divide and Conquer –** I will split up the development of my game into the different scenes I need to create such as the menu screen and the inventory screen so that I can focus on programming each one at a time. I will also further split up the development of each scene into the different classes each one requires such as in the dungeon scene I will need different classes for generating the dungeon and controlling the player's character to move through the dungeon and for the player's sight. After creating all the different scenes, I can combine them to form my complete game and test my whole game as one application.

- **Abstraction –** The user interface of my solution will use simple 2D pixel graphics to display the world which will be abstracted images of the features in my game such as the dungeon and the player character. A game is an abstraction of life for example, when battling enemies, the player loses a 'life' if they get hit rather than their character getting a wound on their leg for example as this is unnecessary information for the player because the player just needs to know how close they are to dying, not the specifics of the injury.

Limitations of my solution

I will be limited by the hardware of the phones I am creating the game for because smart phones are generally less powerful than a normal desktop computer because their CPUs have lower clock times and they have less RAM as well as less secondary storage to store the game on the phone so my app will aim to have as small a file size as possible and also not require too much of the phones processing resources so it will need to be as efficient as possible. Phones are also powered by their battery, so my App should try to use as little battery power as possible while running so that the person using the App can use it throughout the day without their phone running out of power.

Requirements of my solution

Hardware and software requirements

I am choosing to make my app for android phones so anyone who wants to use my app must have an Android phone they could also run the game on their computer as an executable file, but the game will still keep the same screen size as a phone. Using Unity, it is very easy to make copies of my game built for different platforms. I chose to make my app for android phones because designing my App with one platform in mind is easier than designing for multiple platforms, for example,

designing for Apple phones as well as Android while creating my game would mean I would have to keep in mind the differences in hardware of each platform and make adjustments to my game based on the platform it would be running on or possibly create two versions to run on the different platforms. I also helps that I own an Android phone myself that I can use to help test my game.

I will use the game engine, Unity 2017 to help me create my game and the user interface and I will be programming my game in C# using the IDE MonoDevelop. I will use Unity because it is free to use, and I know how to use Unity to create 2D games. Unity also allows you to create games for Android and has a testing feature that allows you to test the game on your phone by connecting it to your computer. I will be using the software, Clip Studio Paint Pro to create the 2D graphics for my game. The requirements below are the system requirements specified on Unity's website that I need to develop the game and run the game I make on an Android phone.

Unity 2017 system requirements to develop my game:

OS: Windows 7 SP1+, 8, 10, 64-bit versions only; Mac OS X 10.9+.

CPU: SSE2 instruction set support.

GPU: Graphics card with DX9 (shader model 3.0) or DX11 with feature level 9.3 capabilities.

Unity system requirements for running the game on Android phones:

OS: Android 4.1 or later.

CPU: ARMv7 CPU with NEON support or Atom CPU.

GPU: OpenGL ES 2.0 or later.

Success Criteria of my solution

My solution will be successful if my stakeholders who test my App, would use my App as their preferred way to practice their mental maths skills before their exams. It will also need to be intuitive to use so that the user can navigate around the App easily and be able to use the App to practice their mental maths and also allow the user to view their progress so that the users can also see if they are improving. My app must also procedurally generate dungeons that are fully working and connected and are different each play through. It must also generate maths questions for the player that depend on the settings they chose and that are different each time.

Interface success criteria

	Criteria	Justification	Evidence
1.1	The Start Menu has a menu with the buttons: new game, continue game and view high scores.	The user is able to choose whether they want to view their high scores, start a new game or continue their current game when the game is loaded.	Screenshot of the first screen when the game is loaded.
1.2	The first three scenes have headings differentiating them	This is so the user knows what scene they are viewing.	Screenshot of the first 3 scenes with headings
1.3	My game uses simple 2D graphics to display the dungeon and character.	The tiles of the dungeon need to be displayed so the player knows where to walk. The	Screenshot of the dungeon scene showing the player character in the dungeon.

		player needs to be displayed so the user can see where their player is on the map.	
1.4	All buttons and input fields have to be able to be interacted with using touch controls.	This is because my game will be a phone app which has so mouse or keyboard like a PC game.	Screenshot of buttons being used from touch controls.
1.5	The player is able to touch the tile they want the player to move to and their character will move to the corresponding location in the dungeon.	Because a phone has no physical keyboard with arrow keys, all movement must be able to be communicated to the player character through touch.	Screenshot of the player character moving to the position touched on screen.
1.6	The player will be able to view their character and the monster they are battling when they encounter a monster.	This is so the player can see what kind of monster they are battling.	Screenshot of the player in a battle with a monster.
1.7	The player will be able to view their character's health and number of potions while on the dungeon or battle screen.	This is so the player can see how close they are to dying and how many potions they have to heal themselves.	Screenshots of the dungeon and battle screens with the player's health and potions shown.
1.8	The player will be able to view the starting number and target number of each maths question.	This is so the player can see what number they are starting with and the number they have to reach and start thinking about how they will solve the question.	Screenshots of the battle screen with the player's starting and target number shown.
1.9	The player will be able to view and select the operations they want to use to solve the maths question.	This is so the player can a simple way of solving the questions and can play just by tapping the buttons.	Screenshots of the battle screen with the operation buttons shown.
1.10	The player will be able to view the amount of moves they have to solve a question which will decrease every time the user picks an operation button.	This is so the player knows how many buttons they have to press to answer a maths question.	Screenshots of the battle screen with the amount of moves left shown.

Functionality success criteria

	Criteria	Justification	Evidence
2.1	Start Menu is the first screen to load	This scene must be the first to load so that the user can navigate to the rest of the game.	Screenshot of the first screen when the game is loaded.
2.2	The continue game button loads the main dungeon screen with the player's last saved game.	This is so the user can immediately continue their game after loading the app.	Screenshot of the start menu with a continue game button.
2.3	The high scores button takes the player to the high scores screen.	This is so the user can check their high scores from the start menu.	Screenshot of the start menu with a high scores button.
2.4	The new game screen loads when the user chooses the new game button from the start menu.	So that the user can choose the settings for their game.	Screenshot of the new game screen with options and buttons to start the game and go back to the start menu.
2.5	When on the high scores scene, the high scores are loaded from a save file and displayed to the user.	This is so the user can view their progress and their best performances.	Screenshot of the high scores screen with the saved high scores viewed.
2.6	The high scores of the game need to be saved.	This is so the user can keep track of their goals and keep trying to get better scores.	Screenshot of the high scores screen with the saved high scores viewed.
2.7	The dungeons are randomly generated.	This is so the player can't get a better score by memorising the map and it means the dungeon levels are infinite, so the user can play the game as much as they want to without playing on the same dungeon and getting bored.	Screenshots of randomly generated dungeons.
2.8	All the dungeon levels have an exit that takes the player to the next level of the dungeon.	This is so the player can play through multiple dungeons in one game because, just one dungeon would be explored too quickly, and one massive dungeon will take a lot longer to generate.	Screenshot of a dungeon with as exit.
2.9	The dungeons get bigger and more complex as the level of the dungeon increases	This is so the dungeons get harder to explore and the player stays on them longer, so they do more maths questions.	Screenshot of the later dungeon levels.
2.10	Maths questions are generated based on the settings chosen by the user when they created a new game.	This is so the user can determine the level of difficulty for their own game and chose one at a level they feel comfortable practicing.	Screenshot of a maths question generated.

2.11	With each question, the user will start with a starting number they are given, and they must use the buttons with operations like "+4" to reach a target number which will be the monsters health.	The questions will be randomly generated so the user doesn't run out of questions to practice. They are given multiple operations to make the questions more difficult than just doing a simple sum.	Screenshot of a maths question generated.
2.12	The player will randomly encounter monsters in the dungeon while exploring.	This is so the player doesn't expect when they will have to do mental maths and will make it hard to reach the exit of each level.	Screenshot of the player encountering a monster.
2.13	The player will be able to heal themselves by using a potion if they have one to give them full health.	This is so the player gets more chances to stay alive in the dungeon and the dungeon isn't too hard for the user.	Screenshot of the player healing themselves.
2.14	If the player tries to incorrectly solve a maths question, the monster they are battling will attack them and the player will lose a life.	This is so the player has some type of pressure to answer the questions correctly but not a stressful pressure like a time limit.	Screenshot of the player guessing the incorrect answer.
2.15	The player will die if they lose all their lives and they lose their progress and will have to start a new game.	This is so the game has a penalty for answering too many questions incorrectly.	Screenshot of the player after they have died.
2.16	The player will gain a health potion from defeating a monster.	This is so that the player gets a prize from answering a question correctly which they can use to increase their chance of survival.	Screenshot of the player after they have defeated an enemy.
2.17	There will be a management system in my game.	This is so all the algorithms that make up the program are coordinated properly to work together since there will be many.	Screenshots of the code of management system classes.

Hardware success criteria

	Criteria	**Justification**	**Evidence**
3.1	The game must be able to be run smoothly using the computational power of an average modern Android phone.	The game must not run slow on the more limited processing power of phones otherwise the user will get frustrated trying to play my game.	Process information of the game running smoothly on my phone.
3.2	The game must use as little memory as possible.	This is because of the limited RAM of phones.	Memory information of the game running on my phone.

3.3	The game must run using as little battery power as possible	This is so that the game can be played for longer and doesn't run the battery down of the phone that is using it so the user can continue to use the phone for other daily tasks.	Battery information of the game running on my phone.

Design

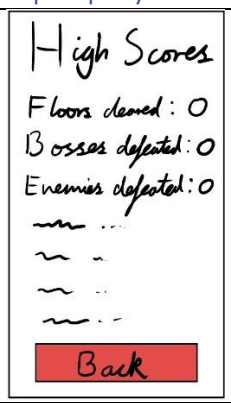
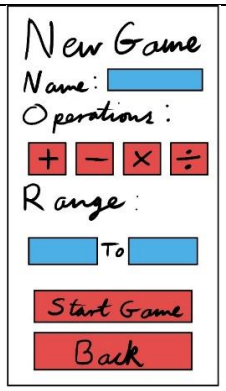
Structure of the solution

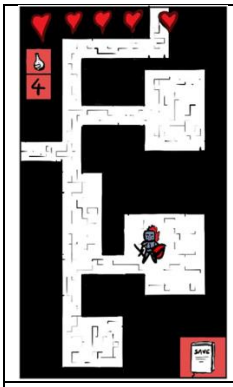
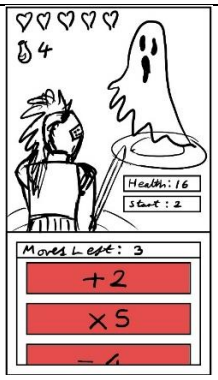
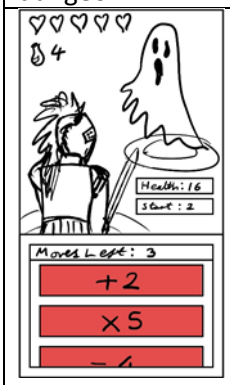
I will be using the Agile design methodology and program using the object-oriented programming paradigm because I will be programming in C# which is built for object-oriented programming. I will break the development of this project into separate scenes, so I will have separate requirements for each scene as well as some components that will be used by all the scenes.

User interface designs

Red areas are interactable buttons, blue areas are interactable input fields where the user can type their input. **Figure 17** is the start menu, **figure 18** is the High Scores scene and **figure 19** is the New Game scene. On the next page, **figure 20** is the dungeon scene, **figure 21** is the first menu of the battle scene and **figure 22** (**all on page 20**) is the second menu of the battle scene after the player picks fight. This is a storyboard of an example playthrough of my game.

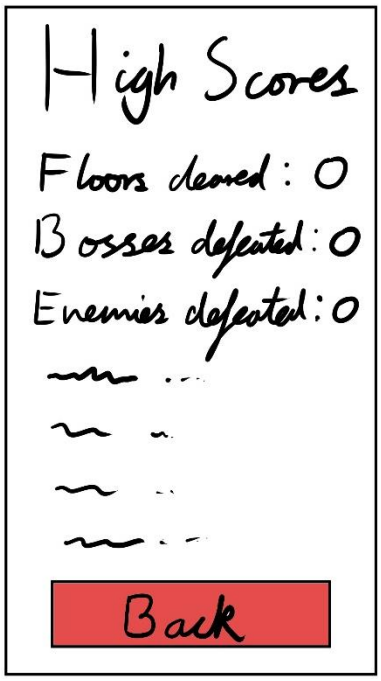
Storyboard of an example playthrough

			
The player loads the game and the menu is loaded. The player selects the High Scores button to view their high scores.	The player's high scores are all zero as this is the first time they have played the game, so they press the back button to return to the menu.	The player is back on the menu and selects new game to create a new game.	The player is now on the new game screen where they enter a name for their character. They pick the operations they want to use for their game. They choose all of them. They pick a range of values from 1 to 50 which will determine the range of numbers that can be the target of a maths question. The player presses start game to start playing.

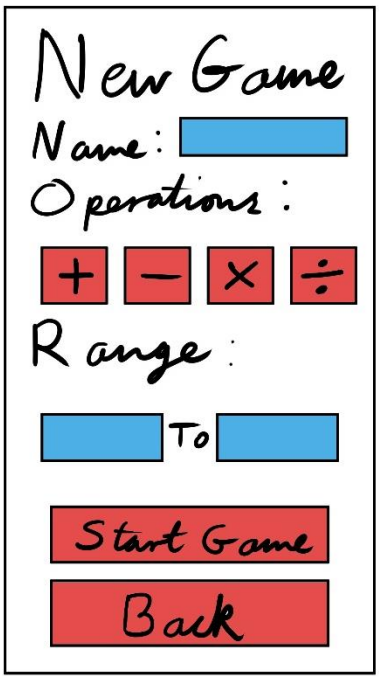
			
The player spawns in a random position in the randomly generated dungeon. They have full health and the starting number of potions. The player selects different areas in the dungeon by touching them to travel there and explore the dungeon.	The player encounters a monster to battle. The monster has 16 health and the player has a starting number of 2. The player selects fight from the menu to try and defeat the monster.	The second menu is shown with the operations the player can use to defeat the monster. The player has 3 moves so they select 3 buttons in the order they think will perform the correct calculations to get from the starting number to the monster's health.	The player is incorrect with their calculation, so they lose a life and are back to the first menu. The player selects heal which uses a potion to give the player full health. The player then selects fight to try defeating the monster again.
			
The player enters a different sequence of buttons and this time they are correct, so they defeat the monster and gain one potion from the battle.	The player returns to their previous position in the dungeon. They find the entrance to the next level and the level is reloaded.	A new dungeon is generated that is bigger with the player in a random position. The player decides to save their game by pressing the save game button.	The game returns the start menu where the player can press continue game to load their game or new game to create a new one.

[Figure 17]

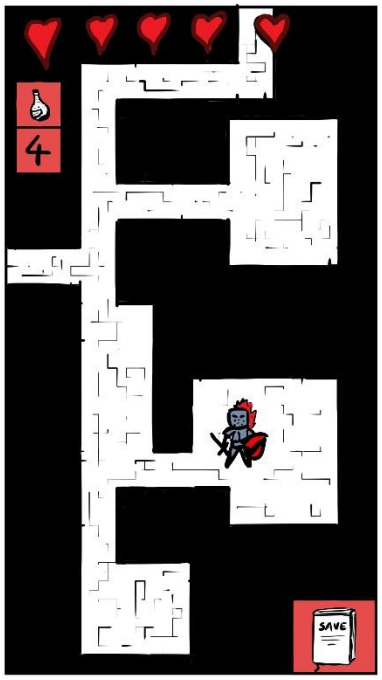
[Figure 18]



[Figure 19]

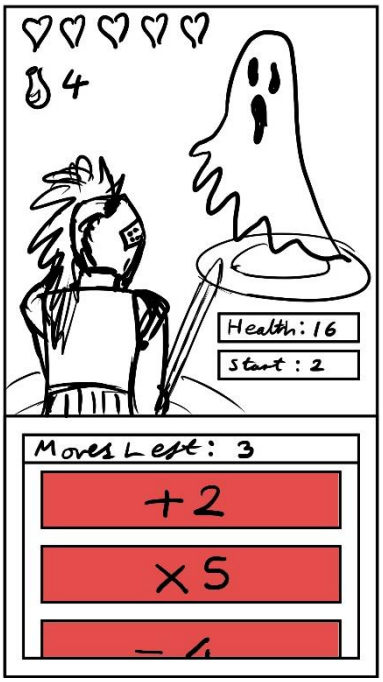


[Figure 20]



[Figure 21]

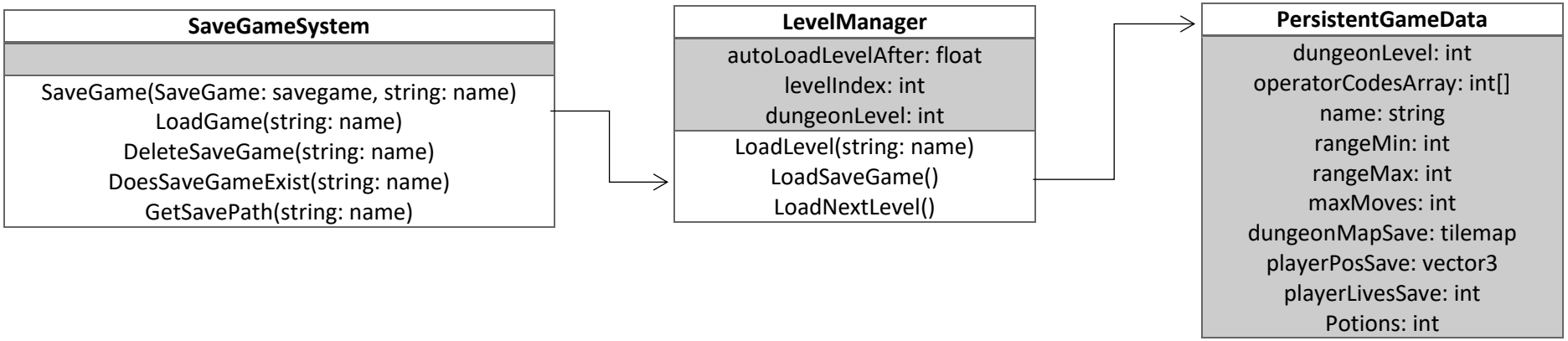
[Figure 22]



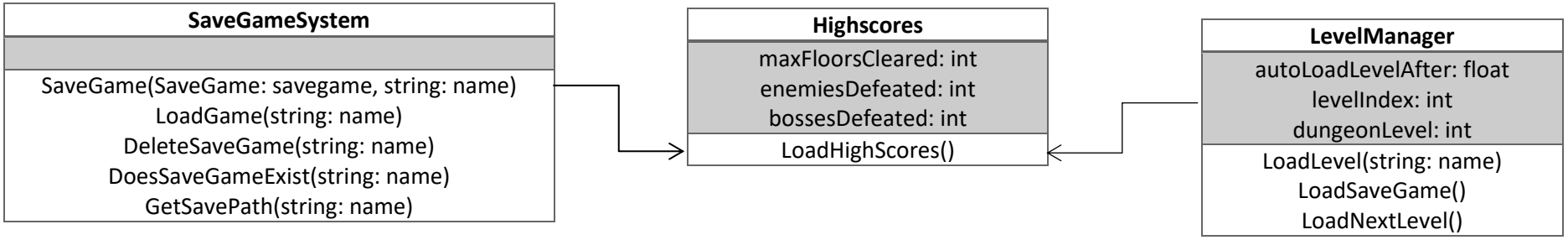
Decomposition of the problem

Class diagrams of the main classes on each scene

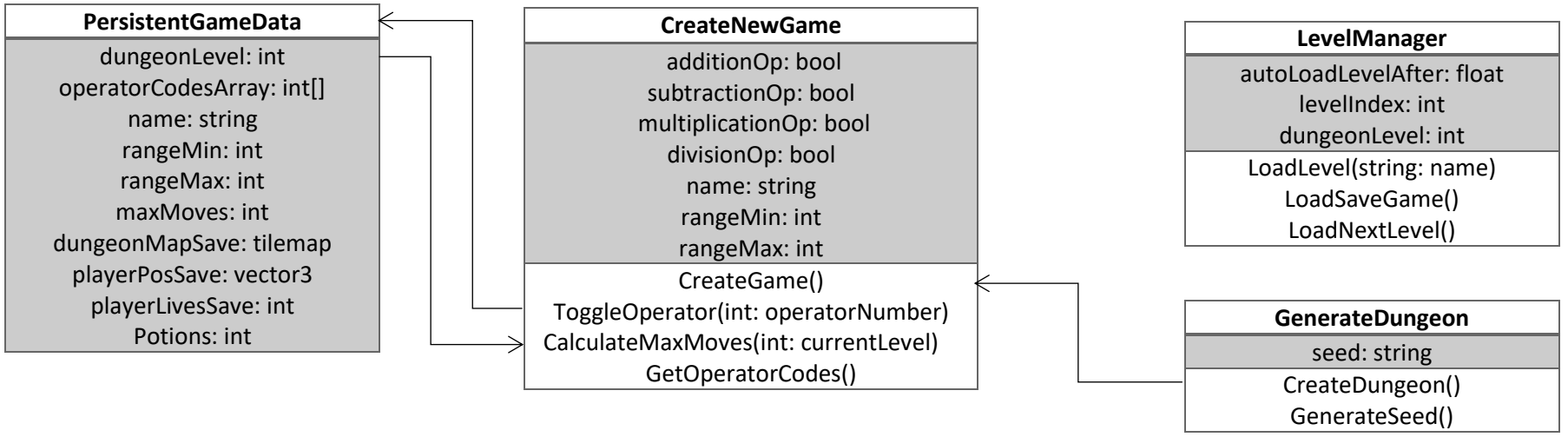
Start Menu scene:



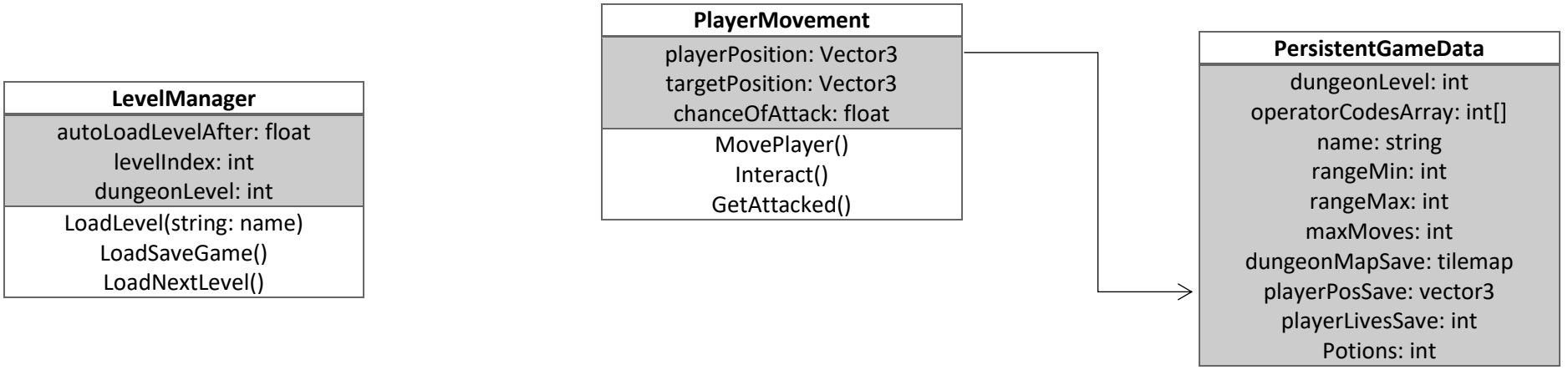
High Scores scene:



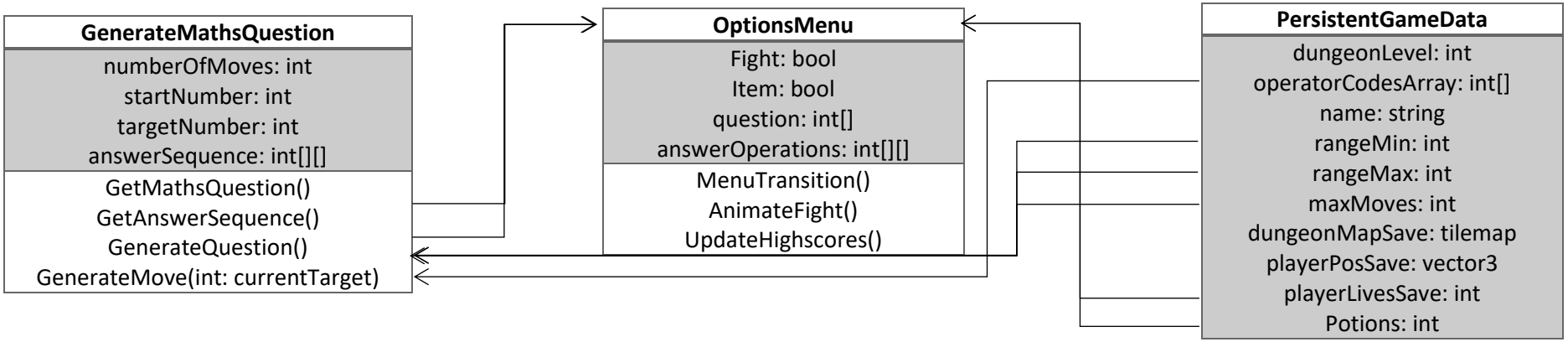
New Game scene:



Main Dungeon scene:



Main Battle scene:



Main Algorithms

Every class in my program inherits from the base class 'MonoBehaviour' (unless stated otherwise) which is a class from the UnityEngine library. It allows the class to use methods from the UnityEngine library such as the Awake, Start and Update methods. Almost all the classes don't use a constructor but instead use the Awake or Start methods or both. These methods, that are inherited from MonoBehaviour, are used to initialise variables before the game starts and are only called once when an instance of a script is being called for the first time. Start is called after Awake on the first frame if the script is enabled. Update is called every frame after Awake and Start have been run.

0) All scenes:

a) Level Manager: The level manager records the current scene index number and can be called by any other class to load a different scene. My game should have 5 scenes in total. It also has an autoload method that will automatically load a level after a certain number of seconds if called. It also has a method that will load the last saved game.

```
PUBLIC CLASS LevelManager
        FLOAT autoLoadLevelAfter = 0
        INT levelIndex

        PRIVATE PersistentGameData pgd
        PRIVATE SaveGameSystem sgs

        //When the start procedure runs, levelIndex will be set to the current
        //scene's index from the build settings. If the autoLoadlevelAfter is not
        //zero, the class will load the next level in the amount of seconds from
        //autoLoadLevelAfter.
        PRIVATE PROCEDURE Start ()
                pgd = PersistentGameData object in scene
                sgs = SaveGameSystem object in scene
                levelIndex = get current level index
                IF autoloadLevelAfter != 0:
                        INVOKE("LoadNextLevel", autoLoadLevelAfter)
                ENDIF
        ENDPROCEDURE

        //This method loads a level with a particular name.
        PROCEDURE LoadLevel (STRING name)
                load level with name: name
        ENDPROCEDURE

        //This method loads the next level in the build settings order.
        PROCEDURE LoadNextLevel ()
                load level with index: levelIndex+1
        ENDPROCEDURE

        //This method finds the last saved game file and loads it.
        PROCEDURE LoadSaveGame ()
                load save file
                assign each variable from the save file to the corresponding
                variables in the persistent data class.
        ENDPROCEDURE
ENDCLASS
```

b) Persistent Game Data: This is the class that will hold all the data that needs to be saved and settings that need to be accessed by all the other classes that are on different scenes. It will be serializable and inherit from the abstract class 'SaveGame'. This class contains no methods except for an 'Awake' method that makes sure this class won't be destroyed.

```
PUBLIC CLASS PersistentGameData INHERITS SaveGame
	INT dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions
	STRING name
	VECTOR3 playerPosSave
	INT[] operatorCodesArray
	TILEMAP dungeonMapSave

	//This stops the game object this class is attached to from being destroyed
	//after it is loaded.
	PRIVATE PROCEDURE Awake ()
		DontDestroyOnLoad(gameObject)
	ENDPROCEDURE
ENDCLASS
```

1) Start Menu scene:

a) Save game system: This class uses binary serialization to save its contents as a permanent file and deserialization to load game data from a permanent file. It has methods to also delete saved games, check if a save game exists and to find the save path. I am using the save game system created by Ryan Nielson which I found online (http://nielson.io/2015/09/saving-games-in-unity/).

2) High scores scene:

a) High scores: All high scores are saved to this class which can be called by options menu class. When on the high scores scene, the load high scores method is called to display all the high scores. This class is serializable so that the high scores can be saved but they will be saved in a separate high scores save file to the saved persistent game data so that the data will not get overwritten when the player creates a new game. The high scores are also from every game played.

```
PUBLIC CLASS HighScores INHERITS SaveGame
	INT maxFloorsCleared
	INT enemiesDefeated
	INT bossesDefeated

	PRIVATE LevelManager levelManager
	PRIVATE SaveGameSystem sgs

	PRIVATE PROCEDURE Awake ()
		DontDestroyOnLoad(gameObject)
	ENDPROCEDURE

	//The level manager object in the scene is found and if the levelIndex is
	//index of the high scores screen, all the scores are loaded.
	PRIVATE PROCEDURE Start ()
		levelManager = levelManager object in scene
		sgs = SaveGameSystem object in scene
		IF levelManager.levelIndex == index of high scores scene:
			LoadHighScores()
		ENDIF
```

```
        //Each high score is loaded into its corresponding textbox on the high
        //scores screen.
        PROCEDURE LoadHighScores ()
                load saved high scores
                assign each variable from the save file to the corresponding
                variables in the persistent data class.
                maxFloorsClearedDisplay.text = maxFloorsCleared
                enemiesDefeatedDisplay.text = enemiesDefeated
                bossesDeafeatedDisplay.text = bossesDefeated
        ENDPROCEDURE
ENDCLASS
```

2. New game scene:

a) Create new game: This class reads all the options the user picks on the new game screen and uses them in the 'create game' method to create a new game with the settings the user has chosen. The settings are passed to the persistent game data class, so that the other classes can read the options the user picks. It also calls the generate dungeon class to generate the map for a new dungeon.

```
PUBLIC CLASS CreateNewGame
        PRIVATE BOOL additionOp, subtractionOp, multiplicationOp, divisionOp
        PRIVATE GenerateDungeon genDungeon
        PRIVATE PersistentGameData pgd

        //Finds the generate dungeon and persistent game data object in the scene.
        PRIVATE PROCEDURE Start ()
                genDungeon = GenerateDungeon object in scene
                pgd = PersistentGameData object in scene
        ENDPROCEDURE

        //When the create new game button is pressed, this method is called and the
        //user's input is read from the text boxes in the create game scene and the
        //generate dungeon object is called to create a new dungeon.
        PROCEDURE CreateGame()
                pgd.name = nameTextbox.text
                pgd.rangeMin = rangeMinTextbox.text
                pgd.rangeMax = rangeMaxTextbox.text
                pgd.maxMoves = CalculateMaxMoves(pgd.dungeonLevel)
                GetOperatorCodes()
                genDungeon.CreateDungeon()
        ENDPROCEDURE

        //This function calculates the maximum amount of moves allowed for each
        //level of the dungeon so each level will get progressively harder as each
        //question can require more moves.
        PRIVATE FUNCTION CalculateMaxMoves(INT currentLevel)
                INT moves = round((4*squareroot(currentLevel))/1.5)
                RETURN moves
        ENDFUNCTION

        //This procedure is called every time the user selects an operator so that
        //they can select or deselect the operations they want to use in a game.
        PROCEDURE ToggleOperator (INT operatorNumber)
                IF (operatorNumber == 1)
                        additionOp != additionOp
                ELSE IF (operatorNumber == 2)
                        subtractionOp != subtractionOp
                ELSE IF (operatorNumber == 3)
                        multiplicationOp != multiplicationOp
```

```
                ELSE
                        divisionOp != divisionOp
                ENDIF
        ENDPROCEDURE

        //Assigns each operator with a code: addition is 1; subtraction is 2;
        //multiplication is 3 and integer division is 4.
        PRIVATE PROCEDURE GetOperatorCodes ()
                INT operatorCode = 0
                IF (additionOp == TRUE)
                        pgd.operatorCodesArray[operatorCode] = 1
                        operatorCode++
                IF (subtractionOp == TRUE)
                        pgd.operatorCodesArray[operatorCode] = 2
                        operatorCode++
                IF (multiplicationOp == TRUE)
                        pgd.operatorCodesArray[operatorCode] = 3
                        operatorCode++
                IF (divisionOp == TRUE)
                        pgd.operatorCodesArray[operatorCode] = 4
                        operatorCode++
        ENDPROCEDURE
ENDCLASS
```

b) Generate dungeon: This class procedurally generates a 2D map of the dungeon that can be then be used to tile the environment. The map should be able to be saved so that it can be loaded when the player wants to continue a saved game. I based my algorithm on the designs by Bob Nystrom (http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/). This class holds an array of tile objects that will be able to be saved. It is made up of many methods for creating the dungeon that have individual jobs:

i) Attempt to place several non-overlapping rooms

ii) Use a maze generator to create the corridors fill in every space around the rooms. I am using a maze generator that is based on Prim's algorithm and learnt from Stack Overflow: https://stackoverflow.com/questions/29739751/implementing-a-randomly-generated-maze-using-prims-algorithm.

iii) Join the corridors and rooms to create one connected area.

iv) Get rid of the dead ends.

This class is the longest, so each method will need to be tested individually before creating the next method to make it easier to check each method is working correctly rather than testing the whole class at the end and finding many errors that are harder to isolate.

c) Code tile: This object is the template for creating tile objects and is what the Generate dungeon class stores an array of. It does not inherit from MonoDevelop as it does not need to access any classes from the Unity Engine library. The tiles have two states: open or closed, they refer to a tile being a walkable floor tile or an unwalkable wall tile. They also have a region number that keeps track of the different regions a tile is part of for example, open tiles that make up one room or connected area will have the same region number.

```
PUBLIC CLASS CodeTile
        ENUMERATION States (Open, Closed)
        STATES myState
        INT regionNumber

        PROCEDURE CodeTile ()
                regionNumber = 0
```

```
                myState = Closed
        ENDPROCEDURE
ENDCLASS
```

3. **Main Dungeon scene:**
   a) **Player movement:** This script records information from the player's touch screen and translates the place they tapped into vector coordinates that the player character will move to. I will use the A* pathfinding algorithm to calculate the path the player character will take through the dungeon because it is the most efficient pathfinding algorithm I know of. It will need to access the array of tiles created by the dungeon generator to know which tiles are able to be walked on.
4. **Main battle scene:**
   a) **Options menu:** This class controls the menu in the battle screen and controls what buttons it displays as well as which are pressed. It records the sequence of buttons the user presses when they try to answer a question. If the user answers correctly, the monster is defeated, otherwise the player misses, and the monster takes a life from the player. If the player loses all their lives, they will die and have to start a new game.

```
PUBLIC CLASS OptionsMenu
        INT[] question
        INT[][] answerOperations
        BOOL fight
        BOOL item

        PRIVATE GenerateMathQuestion genMathQ
        PRIVATE HighScores highScores

        //Finds the generate math question object on the same game object and uses
        //it to get a maths question.
        PRIVATE PROCEDURE Start ()
                highScores = HighScores object in scene
                genMathQ = GenerateMathQuestion object on game object
                question = genMathQ.GetMathsQuestion()
                answerOperations = genMathQ.GetAnswerSequence()
        ENDPROCEDURE

        //Continually checks the state of the player to determine what to do each
        //turn of the battle.
        PRIVATE PROCEDURE Update ()
                IF playerHealth == 0
                        Game over
                ELSE IF playerAnswersCorrect
                        Potions += 1
                        Return to dungeon scene
                ELSE IF playerAnswersIncorrect
                        PlayerLives -= 1
                ENDIF
        ENDPROCEDURE

ENDCLASS
```

   b) **Generate Maths Questions:** This class randomly generates a question at the start of every battle using the user's settings that they chose when they created the game. It will generate a target number and a couple of operations used to get that number which it will pass to the options menu class to display.

```
PUBLIC CLASS GenerateMathQuestion
	INT numberOfMoves
	INT startNumber
	INT targetNumber
	INT[][] answerSequence

	PRIVATE PersistentGameData pgd

	//Finds the create new game object in the scene.
	PRIVATE PROCEDURE Start ()
		pgd = PersistentGameData object in scene
	ENDPROCEDURE

	//This function is called by the options menu class to get the number of
	//moves, the target number and the starting number of the question this
	//class generates.
	FUNCTION GetMathsQuestion ()
		GenerateQuestion()
		RETURN numberOfMoves, targetNumber, startNumber
	ENDFUNCTION

	//This function is called by the options class menu to get the answer
	//sequence of the question generated.
	FUNCTION GetAnswerSequence ()
		RETURN answerSequence
	ENDFUNCTION

	//This procedure performs all the required steps to generate a maths
	//question such as generating a target number and the number of moves
	//the question will take to solve. It generates each move and adds
	//them to a sequence to be passed to the options class.
	PRIVATE PROCEDURE GenerateQuestion ()
		GetOperatorCodes()
		targetNumber = random.range(pgd.rangeMin, pgd.rangeMax)
		numberOfMoves = random.range(1, pgd.maxMoves)
		startNumber = targetNumber
		INT moveNumber = 0
		WHILE (moveNumber != numberOfMoves)
			INT[] moveGenerated = GenerateMove(startNumber)
			answerSequence[moveNumber][0] = moveGenerated[0]
			answerSequence[moveNumber][1] = moveGenerated[1]
			moveNumber++
		ENDWHILE
	ENDPROCEDURE

	//This function generates a move by generating an operator and an integer
	//that the operation will be performed on. The start number is modified by
	//performing the opposite operator on it. When the operator chosen is
	//multiplication, the operator number can only be a factor of the current
	//target to keep all the sums using integer values.
	PRIVATE FUNCTION GenerateMove (INT currentTarget)
		INT operatorsNumber = random.range(0, currentTarget)
		INT operator = random.range(0, pgd.operatorCodesArray.length)
		IF (pgd.operatorCodesArray[operator] == 1)
			startNumber = targetNumber – currentTarget
		ELSE IF (pgd.operatorCodesArray[operator] == 2)
			startNumber = targetNumber + currentTarget
		ELSE IF (pgd.operatorCodesArray[operator] == 4)
			startNumber = targetNumber * currentTarget
		ELSE
			WHILE ((operatorsNumber MOD currentTarget) != 0)
```

```
                        operatorsNumber = random.range(0, currentTarget)
                    ENDWHILE
                    startNumber = targetNumber / currentTarget
                ENDIF
                RETURN pgd.operatorCodesArray[operator], operatorsNumber
            ENDFUNCTION
        ENDCLASS
```

Usability features

The user will tap the screen to interact with buttons and the dungeon map. I will implement simple one-tap gestures for nearly all of the interactive functions of my game. The user will tap buttons with icons to control the app such as touching a bag icon on the main dungeon screen to load the inventory screen. They will also tap the area of the screen corresponding to the area on the dungeon map they want their character to move to. I will also implement a simple pinch-zoom gesture with two fingers, so the user can zoom in and out of the map to see more or less of it.

Key variables and structures

The key variables will depend on which classes have been instantiated on each scene; however the class persistent game data will hold the key variables that all the scenes need to access.

Key variable	Explanation
Dungeon level	Used to determine the size and complexity of the dungeon as well as increase the difficulty of the maths questions. This is increased when the player reaches the exit of the current dungeon level.
Minimum range	The minimum value the target number (monster health) can be for maths questions generated.
Maximum range	The maximum value the target number (monster health) can be for maths questions generated.
Maximum moves	The maximum number of moves a maths question takes to solve. This is increased at a rate proportional to the dungeon level.
Player lives	The number of lives the player currently has. It will have a maximum value of 3. If the player has no lives left, the game ends and the player has to start a new game from the beginning.
Player name	The name of the player's character that will be displayed while battling monsters.
Player position	This variable will save the value of the player character's position in the dungeon so that the player starts a game where they last saved. It will be stored as vector coordinates.
Potions	Stores the number of potions the player can use to heal themselves.
Dungeon map	This will be a grid that stores a map of the dungeon the player is currently exploring. It will contain the positions of the walls and floors and the exit to the next level of the dungeon.

Operator codes array	Stores an array of the numerical codes of the operations (addition, subtraction, multiplication and division) that the user has chosen to use in their game.

The persistent game data class is persistent so that key data is not lost when a new scene is loaded. This class is also serializable so that the data can be saved and reloaded from a permanent file when the player starts the game again. When the player starts a new game, they should always start on the first level of the dungeon with 3 lives and 3 potions.

Validation needs to be done on the new game screen when the user decides the settings for their game. All the input boxes need to be checked first to make sure that the player has typed something in all of them because the player can't start the game without giving their character a name and the range of values they want to use. The name also has to be checked so that it is under the character limit which I will set to 20 characters. The range values need to be checked to make sure the user has only entered integers. The minimum value needs to be checked so that it is greater than 0 and the maximum value should be less than 100000. Both values should be checked to make sure that the minimum value is less than the maximum value. The operations the user has picked should also be checked to make sure the user picked at least one operation they would like to use. An error message should be shown to the user that is specific to the error if any of these validation checks have failed and a new game should not be created.

Test data for development

I will test each class after it has been developed and then test the whole scene after the classes on each scene have been developed. I will test each class to find any syntax errors first and then I will test each class for logic errors. Each class is very different, so the logic errors will depend on what the class should be doing. The classes that save data to a binary serialised file will have to be tested to make sure they hold their data by giving them test data to save, closing the game and checking if the same data can be loaded when the game is run again. Classes that read the user's input will need to be checked that they only allow input that has been validated correctly so I will probably use test tables to keep track of the tests I do and test my validation works. I will test different situations such as when data has not been entered, it is outside an accepted range, it is on the edges of the accepted range and when the correct data is entered.

Test data for beta testing

When development of my game has been completed, I will test the whole game myself to make sure that the whole game runs smoothly. I will test the game on my computer first and once that works I will test it using my phone to make sure it runs on Android devices properly. I will then get my stakeholders to test it using my phone or their own phones if their phone is an android phone. I cannot control how they will play my game, so I hope they will expose any errors I could not find so that I will be able to fix them. They will test the usability of my game to see how easy it is to use and how intuitive the controls are. I will use a questionnaire to record their responses to my questions. I will ask my stakeholders about how easy the game is to use as well as any features that they would like to add to improve the game. I will ask them if they would prefer to use my app than any other method of practicing mental maths. If the majority of my stakeholders prefer to use my game, then my app will be a successful and fun way of helping maths students practice their mental maths.

Review with stakeholders

I showed my interface designs and the storyboard of a playthrough of the game to my stakeholders and asked them a few questions about it to make sure the designs suited their needs. I asked them these questions and recorded their responses:

1. Is the interface clear and easy to understand?
2. Do the controls make sense to you when the game is played on a phone?
3. Do you like the fantasy style of the game?
4. Do you like the system used to answer the maths questions?

Stakeholder comments:

1. "The buttons used to navigate the menu are clear."
   "I didn't understand what the range was for until you explained it. You should add a page of instructions so that people can play the game without you having to explain it."
2. "Letting the player tap the screen is a convenient way of moving around the dungeon."
   "The buttons look easy to tap using a phone."
3. "The fantasy style of style of the game is cool."
   "I like the main character being a knight."
4. "I like the way you have multiple ways of answering the questions as it is different from just having multiple sums to complete and you can combine different operations in one question."

Overall, they liked the interface designs and the fantasy setting for my game. They liked the system used to answer questions as it was different to just giving the user a simple sum to solve which can get boring. To improve my designs, I should add a screen with the instructions or rules of my game as it is not completely obvious from the labels of inputs what should be entered especially the range input boxes on the create new game screen as it is not clear exactly what the range is for. During development I will implement these suggestions.

Development and Testing

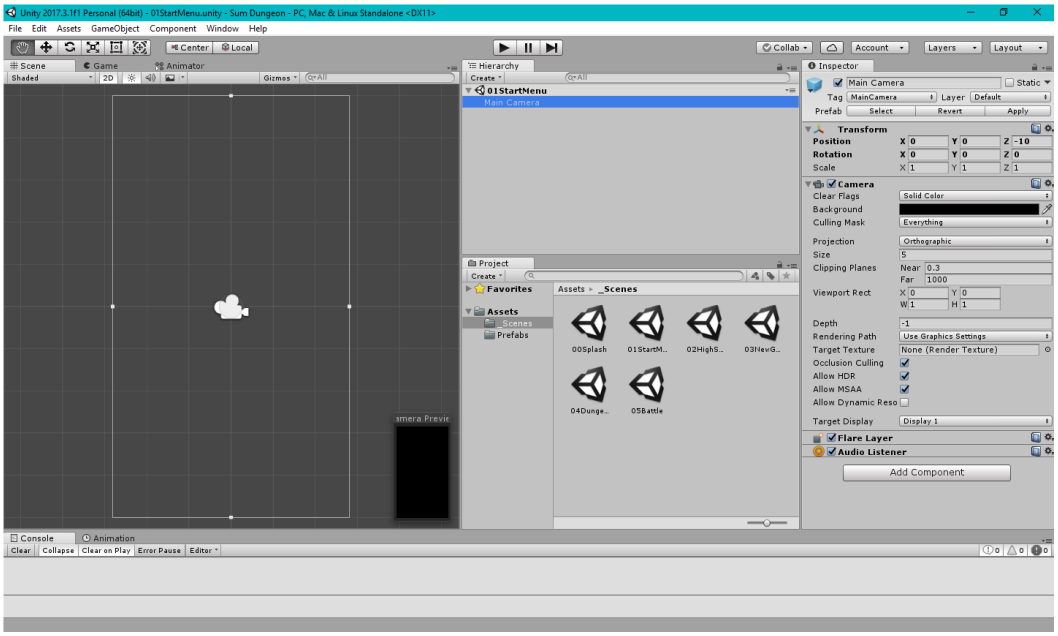
Setting up my game in Unity

Before starting to code my application, I set up all the scenes in the editor of Unity, I gave each scene a number prefixing the name so that the scenes were ordered according to my order of which I expect to be loaded first. I also created a splash scene to be the first scene, which will be a blank scene that autoloads the start menu and is where the classes that create persistent objects are instantiated. This scene cannot be accessed again after the game loads. This is so that the persistent objects are only loaded once, and no duplicates are created from the user going back to this scene where these objects are instantiated. I added all the scenes to the build settings (**Figure 23, page 33**) so that when I test my game, other scenes can be loaded when in play mode and each scene has a build index number which the level manager class can use to reference scenes. When I finally build prototypes of the game, the build settings will be used to tell which scene will be loaded first which will be the Splash scene. The scenes are ordered so that the build index number is the same as the number prefix before each scene name, so I don't have to keep referring to the build settings to know what the build index number for each scene is.

[Figure 23]

I then set up the camera of each scene (**Figure 24, page 33**) by changing the aspect ratio to 9:16 which is the same aspect ratio of my phone which I will use to test my game and so that the camera is vertical. I made the camera background black and made it into a prefab so that the same camera is used on each scene. It is an orthographic camera because my game is 2D, so the camera doesn't need to render depth.

[Figure 24]



Designing the user interface of the first 3 scenes

I created the basic layout of the user interface for the first three scenes based on my interface designs in the design section of my project. When I set up the first button on the start menu (**figure 25, page 34**), I made it a prefab so that it would be easier to change and add functionality to all the buttons later and so that they all had the same style. When creating the high scores screen (**figure 26, page 34**), I created labels for all the different types of high score and separate text boxes for the scores which my high score class will be able to access and use to display each respective score. I put '000' in the boxes as a default placeholder. On the new game screen (**figure 27, page 34**), I added input boxes for the user to enter the name of their character and boxes for them to enter the range of values they want to have in their game. I also added checkboxes that the user can toggle, so that they can choose which mathematical operations they want to use in their game.

[Figure 25]

[Figure 26]

[Figure 27]

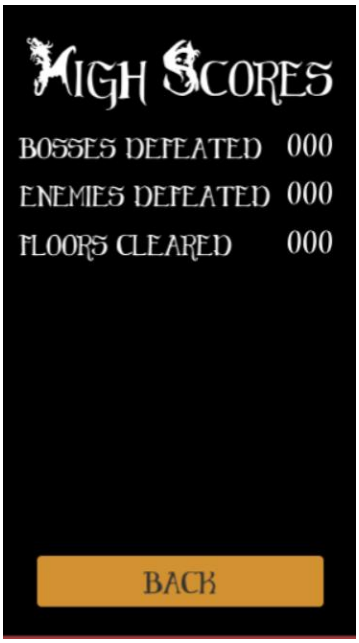
Initial settings review

So far, only the interface success criteria 1.1 has been met and part of 1.2 as I have not started programming, so I have not added any functionality to my game yet. I can only press the buttons and enter text, but they don't do anything yet other than give a visual response because I haven't added any functionality to any of the inputs. I only tested the UI elements to check that I can enter text and that there is a visual response when I press buttons or toggles. When the input boxes are typed in, the text prompt disappears, and I can type whatever I like. The toggles respond by showing a check mark in their box when clicked which disappears if clicked again. The buttons become tinted bronze when pressed. All the buttons from the other scenes work the same as they have been all created from the same prefab, so they work the same apart from label differences. The only change to my initial plans was that I created an extra scene – the splash scene – which doesn't affect my initial designs much. My buttons are shown when they have been pressed in **figures 28, 29 and 30 (all on page 35)**.

[Figure 28]

[Figure 29]

[Figure 30]



Scenes 0 & 1: Splash and Start Menu

Importing the level manager and music manager

The first class I set up was the level manager class (**figure 32, page 36**). I had already written and tested this class when I created it for use in other personal projects, so I imported it into this project and edited it so that it no longer contained code that was specific to the last project I used it for. I also imported a game object called 'fader' which I had created previously to fade the scenes in and out between each transition. It came with its own class (**figure 31, page 35**) that triggers the fade in and out animations and these methods are called by my level manager class when I want a screen to fade to black before loading the next level.

[Figure 31]

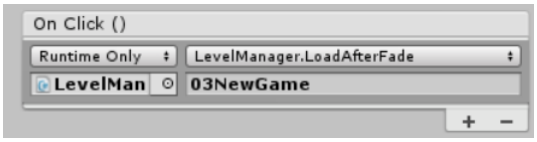
```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class ScreenFader : MonoBehaviour {
6
7     private Animator anim;
8
9     //Initializes the animation component of the fader game object.
10    void Start () {
11        anim = GetComponent<Animator>();
12    }
13
14    // Calls the animator to trigger the fade in sequence of animations.
15    public void FadeIn () {
16        anim.SetTrigger("FadeInTrigger");
17    }
18
19    // Calls the animator to trigger the fade out sequence of animations.
20    public void FadeOut () {
21        anim.SetTrigger("FadeOutTrigger");
22    }
23 }
```

[Figure 32]

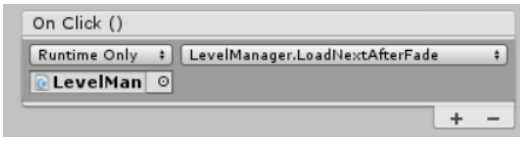
```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;

public class LevelManager : MonoBehaviour {

    public float autoLoadLevelAfter;
    public int currentLoadedSceneIndex;

    private ScreenFader fader;
    private string levelName;

    //When the Awake procedure runs, levelIndex will be set to the current
    //scene's index from the build settings.
    void Awake () {
        currentLoadedSceneIndex = SceneManager.GetActiveScene().buildIndex;
        fader = GameObject.FindObjectOfType<ScreenFader>();
    }

    //If the autoLoadLevelAfter is not zero, the class will load the next level in the amount of seconds from
    //autoLoadLevelAfter.
    void Start () {
        if (autoLoadLevelAfter > 0) {
            Invoke("LoadNextLevel", autoLoadLevelAfter);
        }
    }

     //Loads the level with the name given to it.
     public void LoadLevel(string name){
         Debug.Log ("New Level load: " + name);
         SceneManager.LoadScene(name);
     }

     //Reloads the current level.
     public void ReLoadLevel () {
         SceneManager.LoadScene(currentLoadedSceneIndex);
     }

     //The game stops running and closes.
     public void QuitRequest(){
         Debug.Log ("Quit requested");
         Application.Quit ();
     }

     //Loads the next level in the build settings order.
     public void LoadNextLevel () {
         SceneManager.LoadScene(currentLoadedSceneIndex+1);
     }

     //Tells the fader on the level to fade out and loads the next level in the build settings order after 1 second.
     public void LoadNextAfterFade () {
         fader.FadeOut();
         Invoke("LoadNextLevel", 1f);
     }

    //Tells the fader on the level to fade out and calls the LoadLevelFadeName procedure after 1 second.
    public void LoadAfterFade (string name) {
        fader.FadeOut();
        levelName = name;
        Invoke("LoadLevelFadeName", 1f);
    }

    //Called by the LoadAfterFade procedure to load the level with the name assigned to the variable levelName from
    // LoadAfterFade. This task is split into two prodecures because 'Invoke' can't call a procedure with parameters.
    void LoadLevelFadeName () {
        Debug.Log ("New Level load: " + levelName);
        SceneManager.LoadScene(levelName);
    }
}
```

I added my level manager and fader game objects to each scene, so that when I start developing other scenes I can easily set up the buttons on them to call the level manager's procedures to load other scenes in my game. I connected the buttons to functions from the level manager (**figures 33 and 34, page 37**).

[Figure 33]



[Figure 34]



The New Game button loads the new game scene by passing the name of the scene to the Level manager but the High Scores button just loads the next scene as the high scores scene is the next scene in the build order.

I only added a level manager to the splash scene because this scene is blank and doesn't need a fader. I gave the level manager in this scene an auto load time of one second so that the start menu scene will load automatically after the splash scene. This means that although the start menu scene is not the first scene to load, it will be the first scene the users see as the splash scene is blank, so the users won't notice there was an extra scene before the start menu is loaded. The only method I have not added to the level manager is the LoadSaveGame() method because I have not set up the save game system yet.

Setting up the save system and persistent game data classes

First, I set up the abstract SaveGame class (**figure 35, page 37**) which has no methods as it is a template for any save game type object. This class has a serializable tag so that it can be binary serialised. I made this class inherit from MonoBehaviour instead of not inheriting from any class like in the original designs because I need the classes that inherit from this class (PersistentGameData) to have access to the Unity Engine library so that they can use the Awake and DontDestroyOnLoad functions so that they keep their data while the game is running.

[Figure 35]

```
1 using System;
2 using UnityEngine;
3
4 [Serializable]
5 public abstract class SaveGame : MonoBehaviour {
6     // Save game template.
7 }
```

I then set up the SaveGameSystem (**figure 36, page 38**) class which contains the methods for actually saving and loading games.

[Figure 36]

```
using UnityEngine;
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

public static class SaveGameSystem {

    //Takes a saveGame object and saves it to the file name - 'name'.
    public static bool SaveGame(SaveGame saveGame, string name) {
        //Create a Binary formatter.
        BinaryFormatter formatter = new BinaryFormatter();
        //Create the file stream to where the game will be saved.
        using (FileStream stream = new FileStream(GetSavePath(name), FileMode.Create)) {
            try {
                //Serialise the saveGame object using the binary formatter.
                formatter.Serialize(stream, saveGame);
            }
            catch (Exception) {
                //Returns false if the saveGame object cannot be saved for any reason.
                return false;
            }
        }
        //Returns true if the saveGame object is saved.
        return true;
    }
    //Deserializes the saveGame object with the file name - 'name'.
    public static SaveGame LoadGame(string name) {
        if (!DoesSaveGameExist(name)) {
            //If there is no save with the name - 'name', load nothing.
            return null;
        }
        //Create a binary formatter.
        BinaryFormatter formatter = new BinaryFormatter();
        //Open the file stream to the saveGame file.
        using (FileStream stream = new FileStream(GetSavePath(name), FileMode.Open)) {
            try {
                //Deserialize the file using the binary formatter and return it as a saveGame object.
                return formatter.Deserialize(stream) as SaveGame;
            }
            catch (Exception) {
                //Returns false if the saveGame object cannot be loaded for any reason.
                return null;
            }
        }
    }
    //Deletes a saveGame file with the file name - 'name'.
    public static bool DeleteSaveGame(string name) {
        try {
            //Deletes the file.
            File.Delete(GetSavePath(name));
        }
        catch (Exception) {
            //Returns false if the file cannot be deleted for any reason.
            return false;
        }
        //Returns true if the file is deleted.
        return true;
    }

    //Tries to find the save file with the name - 'name'.
    public static bool DoesSaveGameExist(string name) {
        //Returns true if the file exists otherwise returns false.
        return File.Exists(GetSavePath(name));
    }

    //Finds the file path to the file with the name - 'name'.
    private static string GetSavePath(string name) {
        //Returns a string containing the complete path to the file.
        return Path.Combine(Application.persistentDataPath, name + ".sav");
    }

}
```

I then created the PersistentGameData class (**figure 37, page 39**) to store all the permanent fields in the game. The only field I didn't create was the field to store the map of the dungeon because I wasn't sure what sort of data type I should save it as yet and if the save type would be able to be serialized. I added an annotation to remind myself to come back to this later.

[Figure 37]

```
using System.Collections;
using System.Collections.Generic;
using System;
using UnityEngine;

[Serializable]
public class PersistentGameData : MonoBehaviour {

    public int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
    public string playerName;
    public Vector3 playerPosSave;
    public int[] operatorCodesArray;
    // TODO public TileMap dungeonMapSave

    // This stops the game object this class is attached to from being destroyed when a new scene is loaded.
    void Awake () {
        DontDestroyOnLoad (gameObject);
    }
}
```

Scenes 0 & 1 Review

Success Criteria 2.1, 2.3 and 2.4 have now been met as I can transition between the scenes using the buttons. The Splash scene auto loads the Start Menu scene, so it is the first scene the users can interact with. On the start menu scene, pressing the 'New Game' button, takes you to the new game screen and the 'High Scores' button takes you to the high scores scene. The fader smooths the transition between the scenes by fading out of one scene and fading into the other. The persistent game data is initialised on the Splash scene because it has the 'DontDestroyOnLoad' command. They won't be destroyed when a new scene is loaded and keep their data throughout the game. I have not tested the save game system yet as I want to create the New Game scene first so that I can test the CreateNewGame class with the save system, PersistentGameData and the LevelManager. This is so that I can test all the classes working together to store, load and save the user's data. This section went according to my design plan so there were no changes made to my plan so far.

Scene 2: High scores

I first added the connected the level manager to the back button. In my original designs, the HighScores class stored the high scores data and the methods to display the high scores but I found it better to separate the two, so I created a new class called PersistentHighScores similar to the PersistentGameData class. I separated them so that the HighScores class (**figure 38, page 40**) is only part of the high scores scene but PersistentHighScores (**figure 39, page 40**) is loaded on every scene also, only the high scores data will be serialized instead of a whole class containing methods that are not required to be saved. This means it is also easier to add other types of high scores if I wanted to implement more as I just need to add another variable to the PersistentHighScores class and create more display boxes that the HighScores class can find and use to display the highscore by adding a reference to it in the LoadHighScores() method I created. I decided to use an array to store the display boxes because then it would be simple to just add more identical text boxes with different labels and the HighScores class would find them automatically.

[Figure 38]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class HighScores : MonoBehaviour {

    private GameObject[] textBoxes;
    private Text[] myText;

    //Finds the text boxes that display the high scores and finds the text components of them.
    void Start () {
        //Only the text boxes that are being used to display the high scores have been given the 'DisplayBox' tag.
        textBoxes = GameObject.FindGameObjectsWithTag ("DisplayBox");
        //Loops through each textBox gameObject to find their text component.
        for (int i = 0; i <= (textBoxes.Length-1); i++) {
            myText [i] = textBoxes [i].GetComponent<Text> ();
        }
        //Loads all the high scores after the game objects used to display them have been found.
        LoadHighScores ();
    }
     void LoadHighScores () {
         //Loads the file containing the saved high scores called 'SavedHighScores'.
         PersistentHighScores savedHighScores = SaveGameSystem.LoadGame("SavedHighScores") as PersistentHighScores;
         //Each saved high score gets loaded into it's corresponding text box.
         myText[0].text = savedHighScores.bossesDefeated.ToString();
         myText[1].text = savedHighScores.enemiesDefeated.ToString();
         myText[2].text = savedHighScores.maxFloorsCleared.ToString();
         //When a new high score needs to be added, just use the same format as the lines above but increase the
         //number of myText to access the new display box and reference the specific high score you want to load
         //from the savedHighScores class.
     }
}
```

[Figure 39]

```
using System;
using UnityEngine;

[Serializable]
public class PersistentHighScores : SaveGame {

    public int maxFloorsCleared, enemiesDefeated, bossesDefeated;

    // This stops the game object this class is attached to from being destroyed after it is loaded.
    void Awake () {
        DontDestroyOnLoad (gameObject);
    }
}
```

I attached the PersistentHighScores class to the same game object as the PersistentGameData as both need to be loaded throughout the whole game and have very similar functions.

I tested the HighScores class first by disabling the parts of the program to do with loading the file (**figure 40, page 40**), so I could check the high scores were loaded into the correct display boxes on the user interface. I gave the high scores different test values, so I could make sure each was loaded into the correct display box.

[Figure 40]

```
    void LoadHighScores () {
        //Loads the file containing the saved high scores called 'SavedHighScores'.
        //PersistentHighScores savedHighScores = SaveGameSystem.LoadGame("SavedHighScores") as PersistentHighScores;
        //Each saved high score gets loaded into it's corresponding text box.
        myText[0].text = "1"; //savedHighScores.bossesDefeated.ToString();
        myText[1].text = "2"; //savedHighScores.enemiesDefeated.ToString();
        myText[2].text = "3"; //savedHighScores.maxFloorsCleared.ToString();
        //When a new high score needs to be added, just use the same format as the lines above but increase the
        //number of myText to access the new display box and reference the specific high score you want to load
        //from the savedHighScores class.
    }
```

When I tested this, none of the values were shown because of an error (**figure 41, page 41**) in the Start function.

[Figure 41]

```
6  public class HighScores : MonoBehaviour {
7
8      private GameObject[] textBoxes;
9      private Text[] myText; Field 'HighScores.myText' is never assigned to, and will always have its default value null
10
11     //Finds the text boxes that display the high scores and finds the text components of them.
12     void Start () {
13         //Only the text boxes that are being used to display the high scores have been given the 'DisplayBox' tag.
14         textBoxes = GameObject.FindGameObjectsWithTag ("DisplayBox");
15         //Loops through each textBox gameObject to find their text component.
16         for (int i = 0; i <= (textBoxes.Length-1); i++) {
17             myText [i] = textBoxes [i].GetComponent<Text> ();
18         }
19         //Loads all the high scores after the game objects used to display them have been found.
20         LoadHighScores ();
21     }
```

I had forgot to initialise the array and its length, so I added an extra line to instantiate it under where I instantiated the textBoxes array (**figure 42, page 41**).

[Figure 42]

```
12     void Start () {
13         //Only the text boxes that are being used to display the high scores have been given the 'DisplayBox' tag.
14         textBoxes = GameObject.FindGameObjectsWithTag ("DisplayBox");
15         myText = new Text[textBoxes.Length];
```

When I ran it again, the scores were displayed as I expected them to (**figure 43, page 41**).

[Figure 43]

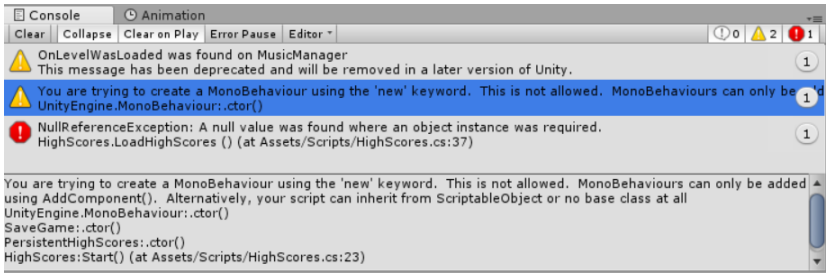
I then moved on to testing loading the high scores from the SavedHighScores file. I added more code to the Start function of the HighScores class to make sure there is always SavedHighScores file that exists using the SaveGameSystem class. I created a new file (**figure 44, page 42**) with the same test values as the previous test. I also uncommented the code that I had taken out for the previous test.

[Figure 44]

```
11      //Finds the text boxes that display the high scores and finds the text components of them.
12      void Start ()
13      {
14          //Only the text boxes that are being used to display the high scores have been given the 'DisplayBox' tag.
15          textBoxes = GameObject.FindGameObjectsWithTag ("DisplayBox");
16          myText = new Text[textBoxes.Length];
17          //Loops through each textBox gameObject to find their text component.
18          for (int i = 0; i <= (textBoxes.Length - 1); i++) {
19              myText [i] = textBoxes [i].GetComponent<Text> ();
20          }
21          //If a high scores file doesn't already exist, create one with all values set to 0.
22          if (!SaveGameSystem.DoesSaveGameExist ("SavedHighScores")) {
23              PersistentHighScores SavedHighScores = new PersistentHighScores();
24              SavedHighScores.bossesDefeated = 1;
25              SavedHighScores.enemiesDefeated = 2;
26              SavedHighScores.maxFloorsCleared = 3;
27              SaveGameSystem.SaveGame(SavedHighScores, "SavedHighScores");
28          }
29          //Loads all the high scores after the game objects used to display them have been found.
30          LoadHighScores ();
31      }
```

Unfortunately, when I ran the code a second time I found out I couldn't have the SaveGame class inherit from MonoBehaviour because classes that inherit from MonoBehaviour can't be initialised using the NEW keyword (**figure 45, page 42**).

[Figure 45]



There were many ways of fixing this, but all would involve changing my plans quite a lot. I would have to change the SaveGame class to work by not inheriting from MonoBehaviour which means the PersistentGameData and PersistentHighScores classes wouldn't be able to use the Awake and DontDestroyOnLoad functions. However, this created the problem of how the data these classes stored could be accessed throughout the game. One way would be to have the PersistentGameData and PersistentHighScores files loaded each time the data was needed from them by another class but that would mean the secondary storage of the device my game would run on would take time to fetch the data each time and different classes could end up with different copies of the data. I thought I would try a different way that still kept the classes being loaded on each scene. Instead of the classes keeping their data between scenes, they could load the data from their own files whenever a new scene was loaded and save it before transitioning to a new scene. This wouldn't change my plans very much and the data would only need to be retrieved from the files once when a scene loads and once before the next scene transition which shouldn't take longer than the time it takes to fade in and out between scenes. Also, the other classes could still access the data and change it quickly if they need to. I used Unity's tutorials to help me change the saving system (https://unity3d.com/learn/tutorials/topics/scripting/persistence-saving-and-loading-data). This gave me the idea to split the PersistentGameData and PersistentHighScores classes into two separate classes each. One which inherited from MonoBehaviour and could be accessed by the other classes to retrieve the data they need. The other would be a private class that only PersistentGameData and PersistentHighScores could access (**figures 48 & 49, page 44**) that inherited from SaveGame and just held the data to be saved to a file each time. The main public class (**figures**

47 & 50, pages 43 and 44 respectively) would use the methods from MonoBehaviour to save and load the data at the right times and store it while on the current scene for the other classes to use. I had to change some of the data types that were saved because I couldn't save data types like Vector3 as they are part of the Unity Engine library and now that SaveGame (**figure 46, page 43**) doesn't inherit from MonoBehaviour, PersistentGameData doesn't recognise these data types. Instead I stored each value of the Vector as its own float variable which is able to be serialised and then I would recreate the original vector from the individual variables when loading the data.

[Figure 46]

```
using System;

[Serializable]
public abstract class SaveGame {
    // Save game template.
}
```

[Figure 47]

```
using System.Collections;
using System.Collections.Generic;
using System;
using UnityEngine;

public class PersistentGameData : MonoBehaviour {

    public int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
    public string playerName;
    public Vector3 playerPosSave;
    public int[] operatorCodesArray;
    // TODO public TileMap dungeonMapSave

    //When this object is instantiated, this class will load the saved data from the file or create a set of new data
    //if there is no file and this is a new game.
    void Awake () {
        bool dataLoaded = LoadGameData();
        if (!dataLoaded) {
            operatorCodesArray = new int[4];
            playerPosSave = new Vector3(0,0,0);
            dungeonLevel = 1;
            playerLivesSave = 3;
            potions = 3;
        }
    }

    //Before this object is destroyed, save the game data to the file.
    void OnDisable () {
        SaveGameData();
    }
    //This method is run before a new scene is loaded to save the data of the previous scene.
    void SaveGameData () {
        GameData SavedGameData = new GameData();
        SavedGameData.dungeonLevel = dungeonLevel;
        SavedGameData.rangeMax = rangeMax;
        SavedGameData.rangeMin = rangeMin;
        SavedGameData.maxMoves = maxMoves;
        SavedGameData.playerLivesSave = playerLivesSave;
        SavedGameData.playerName = playerName;
        SavedGameData.potions = potions;
        SavedGameData.operatorCodesArray = operatorCodesArray;
        SavedGameData.playerPosSaveX = playerPosSave.x;
        SavedGameData.playerPosSaveY = playerPosSave.y;
        SavedGameData.playerPosSaveZ = playerPosSave.z;
        SaveGameSystem.SaveGame(SavedGameData, "SavedGameData");
        Debug.Log("Game data saved.");
    }
```

```
    //This method uses the SaveGameSystem to load the data from the file and returns true if it finds the file.
    bool LoadGameData () {
        if (SaveGameSystem.DoesSaveGameExist ("SavedGameData")) {
            GameData SavedGameData = SaveGameSystem.LoadGame("SavedGameData") as GameData;
            dungeonLevel = SavedGameData.dungeonLevel;
            rangeMax = SavedGameData.rangeMax;
            rangeMin = SavedGameData.rangeMin;
            maxMoves = SavedGameData.maxMoves;
            playerLivesSave = SavedGameData.playerLivesSave;
            playerName = SavedGameData.playerName;
            potions = SavedGameData.potions;
            operatorCodesArray = SavedGameData.operatorCodesArray;

            //The vector of the player's position is created out of the induvidual coordinates for each axis.
            playerPosSave = new Vector3(SavedGameData.playerPosSaveX, SavedGameData.playerPosSaveY, SavedGameData.playerPosSaveZ);
            Debug.Log("Game data loaded.");
            return true;
        }
        return false;
    }
}
```

[Figure 48]

```
//This is a separate private class that inherits from SaveGame and is used by PersistentGameData to store the game data
//in a serialized permentant file.
[Serializable]
class GameData : SaveGame {

    public int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
    public string playerName;
    public float playerPosSaveX, playerPosSaveY, playerPosSaveZ;
    public int[] operatorCodesArray;
    // TODO public TileMap dungeonMapSave

    //The vector of the player's position is saved separately as three floats of each separate coordinate because Vector3
    //is not a standard data type.
}
```

[Figure 49]

```
//This is a separate private class that inherits from SaveGame and is used by PersistentHighScores to store the high score
//data in a serialized permentant file.
[Serializable]
class HighScoreData : SaveGame {

    public int maxFloorsCleared, enemiesDefeated, bossesDefeated;
}
```

[Figure 50]

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PersistentHighScores : MonoBehaviour {

    public int maxFloorsCleared, enemiesDefeated, bossesDefeated;

    //When this object is instantiated, this class will load the saved data from the file or create a set of new data
    //if there is no file and this is a new game.
    void Awake () {
        bool dataLoaded = LoadHighScoreData();
        if (!dataLoaded) {
            maxFloorsCleared = 0;
            enemiesDefeated = 0;
            bossesDefeated = 0;
        }
    }

    //Before this object is destroyed, save the game data to the file.
    void OnDisable () {
        SaveHighScoreData();
    }
```

```
    //This method is run before a new scene is loaded to save the data of the previous scene.
    void SaveHighScoreData () {
        HighScoreData SavedHighScores = new HighScoreData();
        SavedHighScores.maxFloorsCleared = maxFloorsCleared;
        SavedHighScores.enemiesDefeated = enemiesDefeated;
        SavedHighScores.bossesDefeated = bossesDefeated;
        SaveGameSystem.SaveGame(SavedHighScores, "SavedHighScores");
        Debug.Log("High scores saved.");
    }

    //This method uses the SaveGameSystem to load the high score data from the file and returns true if it finds the file.
    bool LoadHighScoreData () {
        if (SaveGameSystem.DoesSaveGameExist ("SavedHighScores")) {
            HighScoreData SavedHighScores = SaveGameSystem.LoadGame("SavedHighScores") as HighScoreData;
            maxFloorsCleared = SavedHighScores.maxFloorsCleared;
            enemiesDefeated = SavedHighScores.enemiesDefeated;
            bossesDefeated = SavedHighScores.bossesDefeated;
            Debug.Log("High score data loaded.");
            return true;
        }
        return false;
    }
}
```

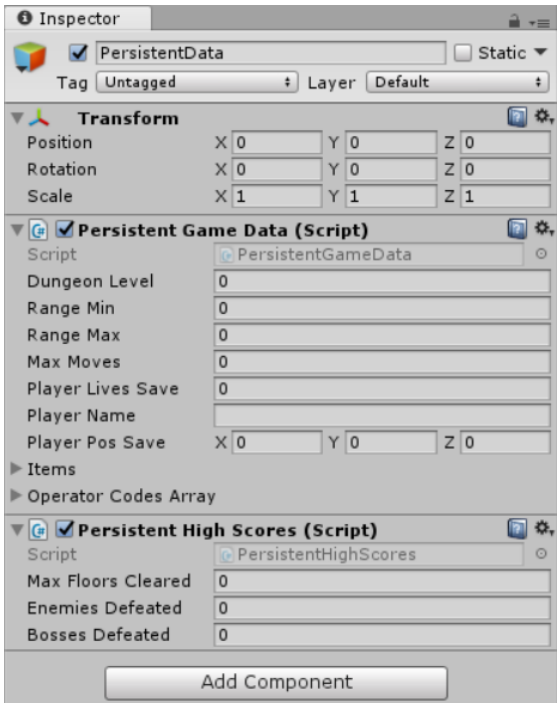
Now all the other classes don't have to load any data from a file, so I edited the HighScores class to just use the variables from the PersistentHighScores class (**figure 51, page 45**).

[Figure 51]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class HighScores : MonoBehaviour {

    private GameObject[] textBoxes;
    private Text[] myText;
    private PersistentHighScores savedHighScores;

    //Finds the text boxes that display the high scores and finds the text components of them as well as the saved high
    //scores.
    void Start () {
        savedHighScores = GameObject.FindObjectOfType<PersistentHighScores>();
        //Only the text boxes that are being used to display the high scores have been given the 'DisplayBox' tag.
        textBoxes = GameObject.FindGameObjectsWithTag ("DisplayBox");
        myText = new Text[textBoxes.Length];
        //Loops through each textBox gameObject to find their text component.
        for (int i = 0; i <= (textBoxes.Length - 1); i++) {
            myText [i] = textBoxes [i].GetComponent<Text> ();
        }
        //Loads all the high scores after the game objects used to display them have been found.
        LoadHighScores ();
    }
     void LoadHighScores () {
         //Each saved high score gets loaded into it's corresponding text box.
         myText[0].text = savedHighScores.bossesDefeated.ToString();
         myText[1].text = savedHighScores.enemiesDefeated.ToString();
         myText[2].text = savedHighScores.maxFloorsCleared.ToString();
         //When a new high score needs to be added, just use the same format as the lines above but increase the
         //number of myText to access the new display box and reference the specific high score you want to load
         //from the savedHighScores class.
     }
}
```

I then created a new game object called PersistentData (**figure 52, page 46**) that the PersistentHighScores and PersistentGameData classes would be attached to, to test that high scores could be loaded and saved to the file. I made it a prefab so that I could add this game object easily to every scene. I changed the default values of the high scores from zero back to the test data I used previously (**figure 53, page 46**) to make sure the correct score was loaded into each Display box.

[Figure 52]

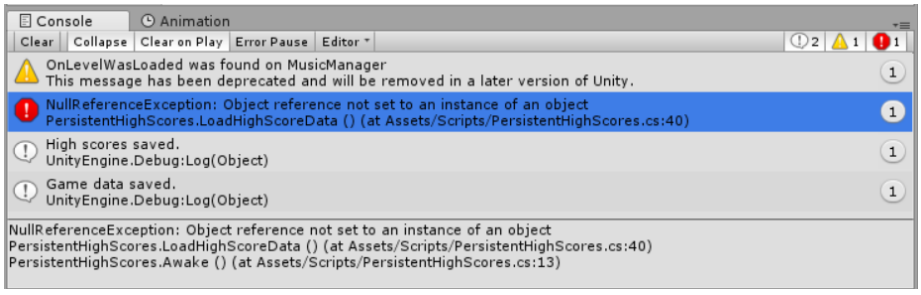


[Figure 53]

```
6 public class PersistentHighScores : MonoBehaviour {
7
8     public int maxFloorsCleared, enemiesDefeated, bossesDefeated;
9
10    //When this object is instantiated, this class will load the saved data from the file or create a set of new data
11    //if there is no file and this is a new game.
12    void Awake () {
13        bool dataLoaded = LoadHighScoreData();
14        if (!dataLoaded) {
15            maxFloorsCleared = 3;
16            enemiesDefeated = 2;
17            bossesDefeated = 1;
18        }
19    }
```

When I tested the high scores scene, the high scores didn't show, and I got an error (**figure 54, page 46**).

[Figure 54]



I did a small test to find out if the variable dataLoaded returned TRUE to see if a SavedHighScores file existed, which it did so there seemed to already be a high scores file. I added some code to change the data of the file (**figure 55, page 47**) to make sure it didn't just display 0 for every high score.

[Figure 55]

```
14    void Start () {
15        savedHighScores = GameObject.FindObjectOfType<PersistentHighScores>();
16        //Only the text boxes that are being used to display the high scores have been given the 'DisplayBox' tag.
17        textBoxes = GameObject.FindGameObjectsWithTag ("DisplayBox");
18        myText = new Text[textBoxes.Length];
19        //Loops through each textBox gameObject to find their text component.
20        for (int i = 0; i <= (textBoxes.Length - 1); i++) {
21            myText [i] = textBoxes [i].GetComponent<Text> ();
22        }
23        //Test data.
24        savedHighScores.bossesDefeated = 1;
25        savedHighScores.enemiesDefeated = 2;
26        savedHighScores.maxFloorsCleared = 3;
27        //Loads all the high scores after the game objects used to display them have been found.
28        LoadHighScores ();
29    }
```

The first test I did was successful as it displayed the new scores (**figure 56, page 47**). I did a second test after taking out the extra code I added before, to make sure the PersistentHighScores class saved the data. The second test was also successful (**figure 57, page 47**) as the high scores were loaded in the exact same way as before. I did a third test to check the data was saved when I switched between scenes by starting from the Splash scene and navigating to the high scores scene to check it loaded the same scores. This test was also successful (**figure 58, page 47**), and I could even go back and forth between the Start Menu scene and the HighScores scene and the data was still saved. I could also see that the data was being loaded and saved from the console in Unity's editor because I added code that printed a message when the data was saved or loaded.

[Figure 56]

[Figure 57]

[Figure 58]

Scene 2 Review

Both success criteria 2.5 and 2.6 have been met as the back button loads the Start Menu screen and the correct high scores are loaded and displayed properly. Although I haven't tested the high scores system by updating them based on the game play, I am confident I will be able to easily implement it, and have it work properly because the other classes just need to refer to the PersistentHighScores class to change the data which will be saved properly. I also tested the PersistentGameData class so that it is able to be used in the same way to load key game data. I will be testing this more in the next scene when data is entered by the player and stored in the PersistentGameData class. I changed my original plans quite a lot during this section, for example the level manager no longer

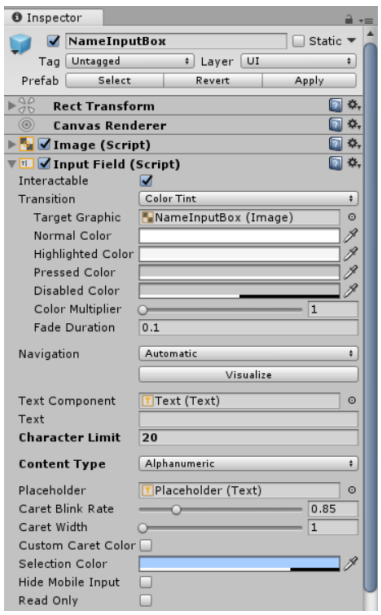
needs a method to or load data as this is done only by the PersistentGameData class now. The persistent classes have been split into two different classes, with one holding the data permanently and the other only holding it while the game is running.

Scene 3: New game

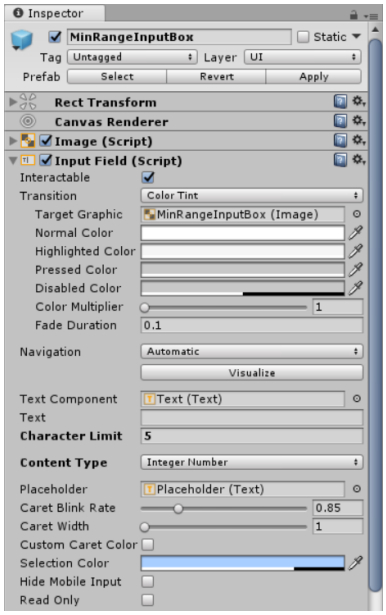
Creating the system to let the player pick the game's settings

The first thing I did was set up the Start Game and Back buttons by attaching them to the level manager. I edited the UI elements I had created when setting up the scenes by adding validation to them which I did in Unity's editor. I changed the content type of the NameInputBox (**figure 59, page 48**) to be alphanumeric because I didn't want the name to have any strange characters in it that might cause problems for my program when reading the name. I also made the character limit 20 characters so that the name won't trail off the screen when displayed. I changed the content types of the MinRangeInputBox (**figure 60, page 48**) and MaxRangeInputBox (**figure 61, page 48**) to be integer number. I also changed the character limit to 5 characters because the player shouldn't be able to enter numbers that are too big and have more than 5 digits.

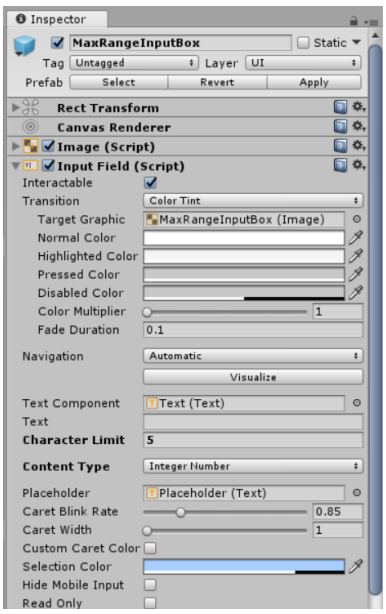
[Figure 59]



[Figure 60]



[Figure 61]



I then created the CreateNewGame class (**figure 62, page 49**). I started creating it according to my design that I wrote in pseudocode, but I realised I could shorten the code by using the classes from the Unity Engine UI library. I removed the Boolean variables and the ToggleOperator method because it was redundant, instead I just found all the toggle game objects in the scene and used their 'isOn' property and checked if it was true to know if the user had picked that operation. I attached the input field objects manually to the class in the editor by making the variables public because then I knew exactly which input field was which. I could just read directly what the input was because the input field objects do a lot of the validation themselves but for the minimum and maximum range I had to add extra validation to make sure they were within the range of values (between 1 and 99999) that I specified in the design section. The input field only allows the player to enter integers, but I don't want them to enter negative numbers and the maximum value must be greater than the minimum value.

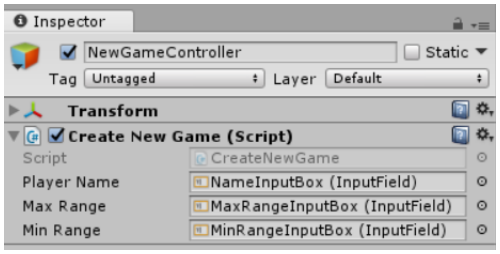
[Figure 62]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class CreateNewGame : MonoBehaviour {

    public InputField playerName, maxRange, minRange;

    private PersistentGameData pgd;
    private Toggle[] myToggles;
    //TODO private GenerateDungeon genDungeon

    //Finds the generate dungeon and persistent game data object in the scene.
    void Start () {
        pgd = GameObject.FindObjectOfType<PersistentGameData> ();
        myToggles = GameObject.FindObjectsOfType<Toggle>();
        //TODO add refernces to the dungeon generator class when made.
    }
    //Called when the player presses the Start Game button and reads what the player has entered.
    public void CreateGame () {
        pgd.playerName = playerName.text;
        //Parsing the text to convert it into integers.
        int workingMin = int.Parse(minRange.text);
        int workingMax = int.Parse(maxRange.text);
        //Min and Max validation.
        if (workingMax > workingMin) {
            if (workingMin > 0 && workingMax < 1000000) {
                pgd.rangeMin = workingMin;
                pgd.rangeMax = workingMax;
            }
        }
        pgd.maxMoves = CalculateMaxMoves (pgd.dungeonLevel);
        GetOperatorCodes();
        //TODO genDungeon.GenerateDungeon(); - from dungeon generator class.
    }

    //A formula I came up with myself to increase the maximum moves for a maths problem as the player progresses.
    int CalculateMaxMoves (int currentLevel) {
        float cLevel = (float)currentLevel;
        float fMoves = (4f * Mathf.Sqrt (cLevel)) / 1.5f;
        int moves = Mathf.RoundToInt(fMoves);
        return moves;
    }
    //The operatorCodesArray stores the different operations the player wants to use, 1 is addition, 2 is
    //subtraction, 3 is multiplication and 4 is division.The different toggles corespond to different
    //operators.
    void GetOperatorCodes () {
        int operatorCode = 0;
        for (int i = 0; i <= 3; i++) {
            if (myToggles[i].isOn == true) {
                pgd.operatorCodesArray [operatorCode] = (i + 1);
                operatorCode++;
            }
        }
    }
}
```

I left out the dungeon generating parts of this class because I haven't created the dungeon generator class yet.

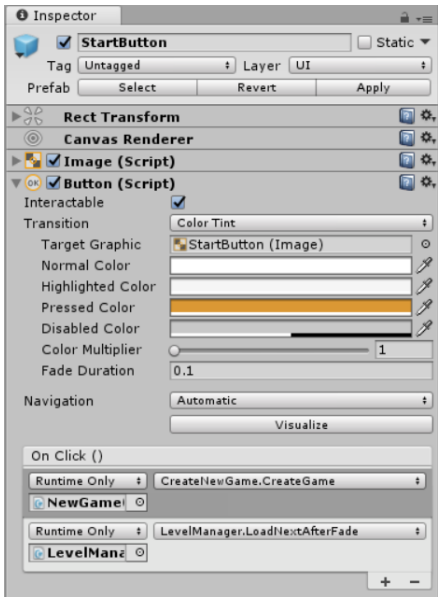
I attached this class to a new game object I made called NewGameController (**figure 63, page 50**) and linked the input fields in the scene to it.

[Figure 63]



I then added functionality to the StartButton (**figure 64, page 50**) so that it calls the CreateGame function from the CreateNewGame class. It will then call the level manager to load the next level which is the dungeon scene and will trigger the PersistentGameData class to save the player's choices to the save game file.

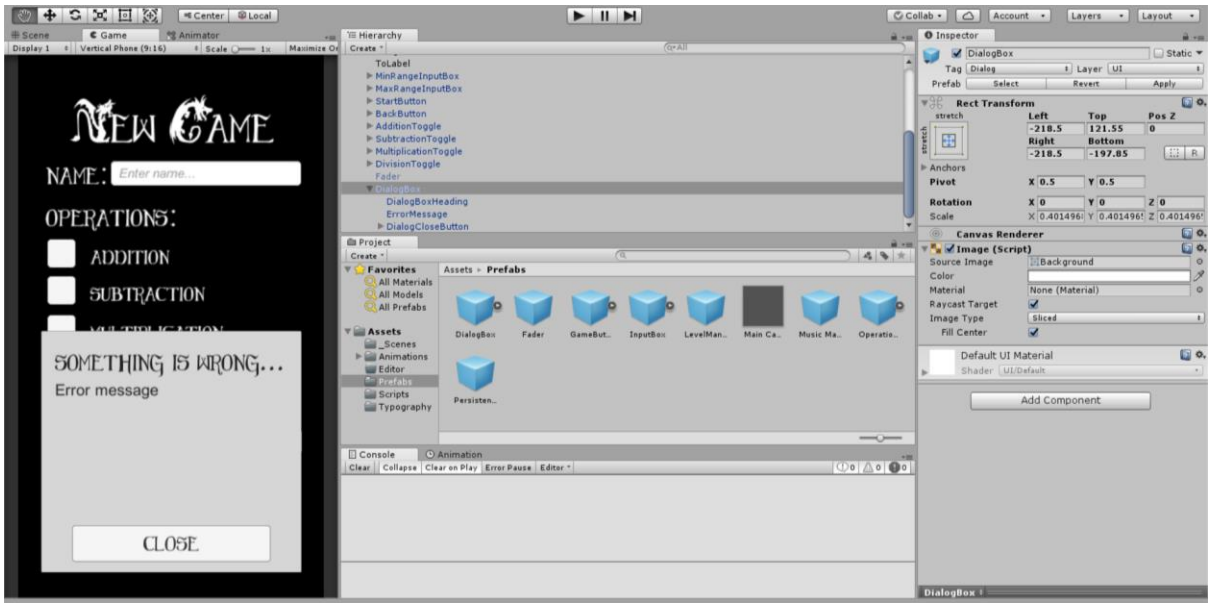
[Figure 64]



Adding the dialog box to display errors to the player

I then wanted to create a dialog box game object (**figure 65, page 51**) that would appear if the user entered the wrong data and told them what they did wrong and wouldn't let them start a new game until all their entries were correct. I created four new game objects to create the dialog box. I first created a panel that would appear over the Start Game and Back buttons to stop the user clicking them again after the dialog box appeared. Then I created a heading for the box and a separate text game object for where the error message would be displayed. I added an 'CLOSE' button that the user should press to get rid of the dialog box. I also created a new tag called dialog which the dialog box would have so it could be found by the CreateNewGame class. I also made the dialog box a prefab, so I could use it in later scenes.

[Figure 65]



I then created a new class called DialogErrorDisplay (**figure 66, page 51**) attached to the error message text of the dialog box. The CreateNewGame class would be able to call a method from this class to display the error messages.

[Figure 66]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class DialogErrorDisplay : MonoBehaviour {

    private Text errorDisplay;

    //Initialises the text component of the game object.
    void Start () {
        errorDisplay = GetComponent<Text>();
    }

    //Called by the CreateNewGame class to pass a message to the dialog box to display.
    public void DisplayError (string errorMessage) {
        errorDisplay.text = errorMessage;
    }
}
```

I also edited the CreateNewGame class (**figure 67, page 52**), so it would be able to find the dialog box game object and the error display to display it when needed. I refactored the code in CreateNewGame so that the code that checked the user's input was in a separate function. This function would only return true if all the user's inputs were correct, otherwise it would display the dialog box with an error message telling them what they did wrong. I removed the second function of the Start Button that loaded the next level because I only wanted the dungeon to load when all the values the player had inputted were correct. I instead, moved the call to load the next scene to inside the CreateNewGame class in the CreateGame method that will still be called when the player presses the Start Game button. I also added validation to the toggles to make sure at least of the operations was selected.

[Figure 67]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class CreateNewGame : MonoBehaviour {

    public InputField playerName, maxRange, minRange;

    private bool validSelection;
    private PersistentGameData pgd;
    private Toggle[] myToggles;
    private GameObject dialogBox;
    private DialogErrorDisplay errorDisplay;
    private LevelManager lvlManager;
    //TODO private GenerateDungeon genDungeon

    //Finds all the classes it needs data from, including the PersistentGameData, all the toggles, the level
    //manager, the dungeon generator, the dialog box and the display to print error messages.
    void Start () {
        pgd = GameObject.FindObjectOfType<PersistentGameData> ();
        myToggles = GameObject.FindObjectsOfType<Toggle>();
        lvlManager = GameObject.FindObjectOfType<LevelManager>();
        dialogBox = GameObject.FindGameObjectWithTag("Dialog");
        errorDisplay = GameObject.FindObjectOfType<DialogErrorDisplay>();
        //TODO find the dungeon generator class when made.
    }
    //Called when the player presses the Start Game button and reads what the player has entered.
    public void CreateGame () {
        bool correctInput = ValidateInputs();
        GetOperatorCodes();
        if (validSelection == false) {
            OpenDialogBox("Please pick at least one operator to use.");
        } else if (correctInput == true) {
            pgd.maxMoves = CalculateMaxMoves (pgd.dungeonLevel);
            //TODO genDungeon.GenerateDungeon(); - from dungeon generator class.
            lvlManager.LoadNextAfterFade();
        }
    }
   //Checks all the user inputs to make sure they are valid.
   bool ValidateInputs () {
       //If the player hasn't entered a name.
       if (playerName.text.Length == 0) {
           OpenDialogBox ("You forgot to enter a name.");
       } else {
           pgd.playerName = playerName.text;
           //Parsing the text to convert it into integers.
           int workingMin;
           int workingMax;
           //Convert the text into integers.
           bool minResult = int.TryParse (minRange.text, out workingMin);
           bool maxResult = int.TryParse (maxRange.text, out workingMax);
           //If the player hasn't entered a range.
           if (minResult == false) {
               OpenDialogBox ("You forgot to enter a minimum value.");
           } else if (maxResult == false) {
               OpenDialogBox ("You forgot to enter a maximum value.");
           } else {
               //If the minimum value is greater than the maximum values.
               if (workingMax < workingMin) {
                   OpenDialogBox ("Your minimum value is greater than your maximum value, please swap the values.");
               } else {
                   //If the minimum value is 0 or negative.
                   if (workingMin < 0) {
                       OpenDialogBox("Your minimum value must be greater than zero.");
                   //If the maximum value is above 100000 which is unreasonable for mental maths.
                   } else if (workingMax > 100000) {
                       OpenDialogBox("Your maximum value must be less than 100000.");
```

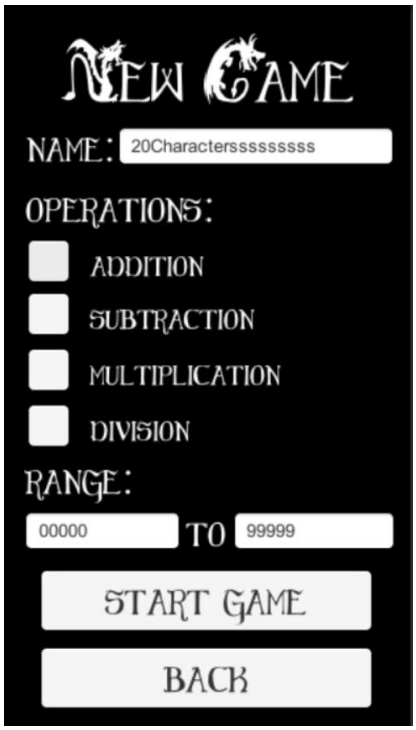
```
71                  } else {
72                      //If all conditions are met, the values are saved.
73                      pgd.rangeMin = workingMin;
74                      pgd.rangeMax = workingMax;
75                      return true;
76                  }
77              }
78          }
79      }
80      return false;
81  }
82
83  //Displays the dialog box with the error message given to it.
84  void OpenDialogBox (string message) {
85      dialogBox.SetActive(true);
86      errorDisplay.DisplayError(message);
87  }
88
89  //Called when the user presses the CLOSE button on the dialog box to stop displaying the box.
90  public void CloseDialogBox () {
91      dialogBox.SetActive(false);
92  }
93
94  //A formula I came up with myself to increase the maximum moves for a maths problem as the player progresses.
95  int CalculateMaxMoves (int currentLevel) {
96      float cLevel = (float)currentLevel;
97      float fMoves = (4f * Mathf.Sqrt (cLevel)) / 1.5f;
98      int moves = Mathf.RoundToInt(fMoves);
99      return moves;
100 }
102  //The operatorCodesArray stores the different operations the player wants to use, 1 is addition, 2 is
103  //subtraction, 3 is multiplication and 4 is division.The different toggles corespond to different
104  //operators.
105  void GetOperatorCodes () {
106      validSelection = false;
107      int operatorCode = 0;
108      for (int i = 0; i <= 3; i++) {
109          if (myToggles[i].isOn == true) {
110              pgd.operatorCodesArray [operatorCode] = (i + 1);
111              operatorCode++;
112              validSelection = true;
113          }
114      }
115  }
116 }
```

I attached the CloseDialogBox function to the close button of the dialog box and disabled the whole dialog box to make sure it only showed up when there was an error.

Testing the validation system

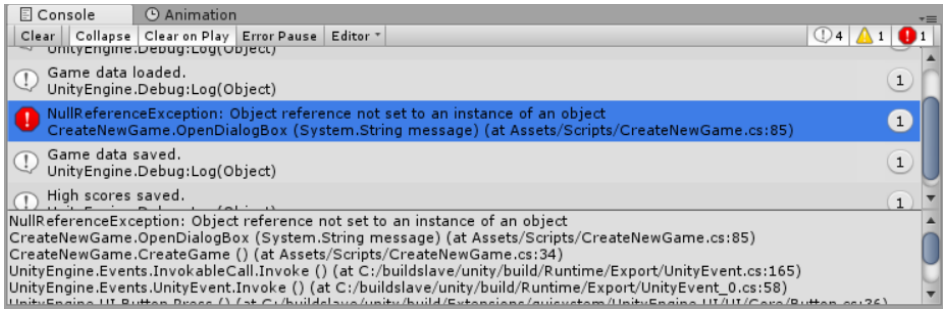
I then started to test the system. First, I made sure to check that the content types I specified for each input field worked. I first tried entering data into the name input field (**figure 68, page 54**), I found I could only type letters or numbers, so the alphanumeric setting worked correctly. I then tested the minimum and maximum value input field, I found I could only enter numbers and only the first character could be a dash (to type negative integers) so the integer setting also worked correctly. I tested the character limit as well to make sure that worked. I could only enter up to 20 characters in the name input field and 5 digits in the minimum and maximum input fields, so the character limit worked.

[Figure 68]



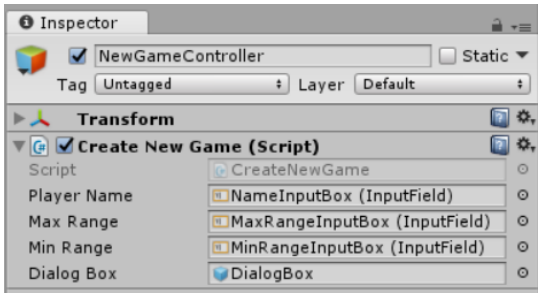
I then wanted to make sure that the player could not start a game without picking one of the operations, so I tried to start the game using valid data but not picking any operations. Unfortunately, nothing happened, and I got an error when pressing the start game button (**figure 69, page 54**).

[Figure 69]



I thought this error was because the CreateNewGame class couldn't find the dialog box and enable it because it was disabled so I edited the class by making the reference to it public (**figure 71, page 55**) and attaching it manually in Unity's editor (**figure 70, page 54**).

[Figure 70]

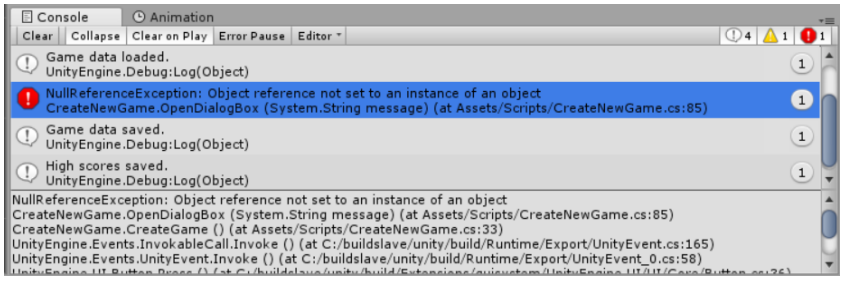


[Figure 71]

```
6 public class CreateNewGame : MonoBehaviour {
7
8     public InputField playerName, maxRange, minRange;
9     public GameObject dialogBox;
10
11    private bool validSelection;
12    private PersistentGameData pgd;
13    private Toggle[] myToggles;
14    private DialogErrorDisplay errorDisplay;
15    private LevelManager lvlManager;
16    //TODO private GenerateDungeon genDungeon
17
18    //Finds all the classes it needs data from, including the PersistentGameData, all the toggles, the level
19    //manager and the dungeon generator.
20    void Start () {
21        pgd = GameObject.FindObjectOfType<PersistentGameData> ();
22        myToggles = GameObject.FindObjectsOfType<Toggle>();
23        lvlManager = GameObject.FindObjectOfType<LevelManager>();
24        errorDisplay = GameObject.FindObjectOfType<DialogErrorDisplay>();
25        //TODO find the dungeon generator class when made.
26    }
```

When I tried the test again I still didn't get a proper error message, but I got a different response. The dialog box appeared, but no error message was displayed (**figure 73, page 55**), and I got the same error from the console (**figure 72, page 55**).

[Figure 72]



[Figure 73]

I was now sure that the CreateNewGame class couldn't find the dialog box object because it was disabled. The same problem was happening to the DialogErrorDisplay class because it was attached to the disabled ErrorMessage game object when the CreateNewGame class was trying to find it. I decided to edit the code of the class again to instead also manually find the DialogErrorDisplay (**figure 74, page 56**) and only when the gameobject it is attached to (**figure 76, page 56**) is active (**figure 75, page 56**).

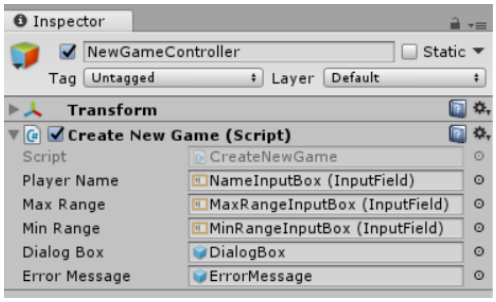
[Figure 74]

```
6 public class CreateNewGame : MonoBehaviour {
7
8     public InputField playerName, maxRange, minRange;
9     public GameObject dialogBox, errorMessage;
```

[Figure 75]

```
81     //Displays the dialog box with the error message given to it.
82     void OpenDialogBox (string message) {
83         dialogBox.SetActive(true);
84         errorDisplay = errorMessage.GetComponent<DialogErrorDisplay>();
85         errorDisplay.DisplayError(message);
86     }
```

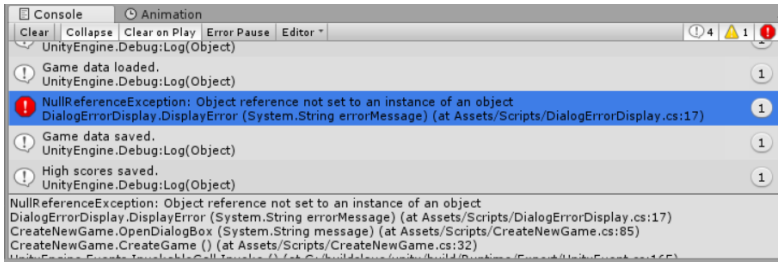
[Figure 76]



When I tested this again I got the same display as the last test (**figure 77, page 56**) but a different error message (**figure 78, page 57**).

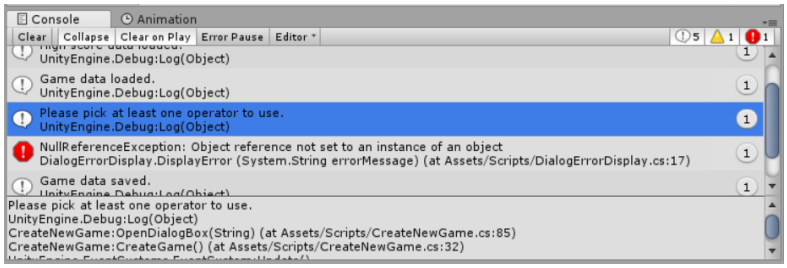
[Figure 77]

[Figure 78]



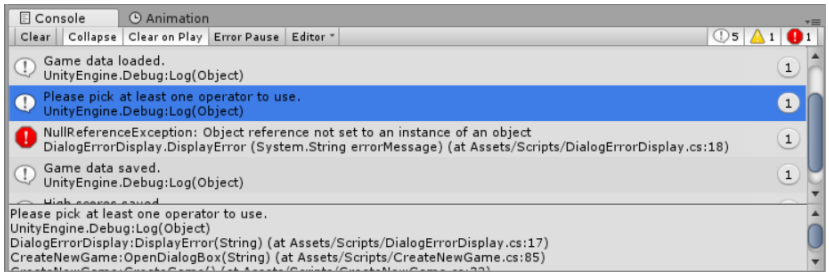
I decided to check what the error message that the DialogErrorDisplay class was trying to display by logging it to the console. I first checked the CreateNewGame class which did have the correct error (**figure 79, page 57**).

[Figure 79]



I then checked the DialogErrorDisplay class which also had the correct error message (**figure 80, page 57**).

[Figure 80]



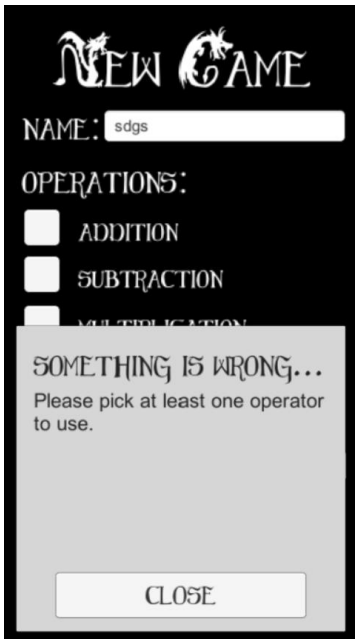
I then realised the problem was that the DialogErrorDisplay class couldn't find its own text component because it was disabled, so I moved the line of code that found the text component to the DisplayError method where I knew the game object would be active when this method was called (**figure 81, page 57**).

[Figure 81]

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5
6  public class DialogErrorDisplay : MonoBehaviour {
7
8      private Text errorDisplay;
9
10     //Called by the CreateNewGame class to initialise the text component of the game object and pass
11     //a message to the dialog box to display.
12     public void DisplayError (string errorMessage) {
13         errorDisplay = GetComponent<Text>();
14         errorDisplay.text = errorMessage;
15     }
16 }
```

This time when I tried the test again I got the correct error message to display (**figure 82, page 58**). I thought the dialog box was too big for the error messages, so I changed the design of it slightly (**figure 83, page 58**). I also tested picking an operator and then unpicking it to make sure the error message displayed (**figure 84, page 58**).

[Figure 82]



[Figure 83]

[Figure 84]

Now that the Dialog box worked, I wanted to test the validation that I had coded. I created a table of different inputs I wanted to try (**figure 85, page 58**) and what I expected the result of each test to be to help me keep track of the tests I had done. In all the tests, I had all the operators selected.

[Figure 85]

Inputs			Expected Output	Actual Output	Successful Test?
Name	**Minimum value**	**Maximum value**			
(nothing)	(nothing)	(nothing)	Missing name error	Missing name error	Yes
(nothing)	1	9	Missing name error	Missing name error	Yes
abc	(nothing)	9	Missing minimum value error	Missing minimum value error	Yes
abc	1	(nothing)	Missing maximum value error	Missing maximum value error	Yes
abc	9	1	Minimum value is greater than maximum error	Minimum value is greater than maximum error	Yes
abc	-2	-6	Minimum value is greater than maximum error	Minimum value is greater than maximum error	Yes

abc	1	-6	Minimum value is greater than maximum error	Minimum value is greater than maximum error	Yes
abc	-2	3	Minimum value must be greater than 0 error	Minimum value must be greater than 0 error	Yes
abc	1	99999	Saves data and loads next scene	Saves data and loads next scene	Yes
abc	0	99999	Minimum value must be greater than 0 error	Saves data and loads next scene	No
abc	-6	-2	Minimum value must be greater than 0 error	Minimum value must be greater than 0 error	Yes
Elzbieta	5	20	Saves data and loads next scene	Saves data and loads next scene	Yes

The only test that failed was the third to last test, so I made a small change to the CreateNewGame class to fix it (**figure 86, page 59**): (The class used to check if workingMin was less than 0).

[Figure 86]

```
} else {
    //If the minimum value is 0 or negative.
    if (workingMin < 1) {
        OpenDialogBox("Your minimum value must be greater than zero.");
```

I did the test again and this time it was successful.

After doing this test I realised that the validation to check if the range was less than 100000 was redundant as I already set the character limit of the input boxes to be 5 characters, so the user can't enter 100000 or any number higher with 6 or more digits. I removed the code to check this from the CreateNewGame.

Scene 3 Review

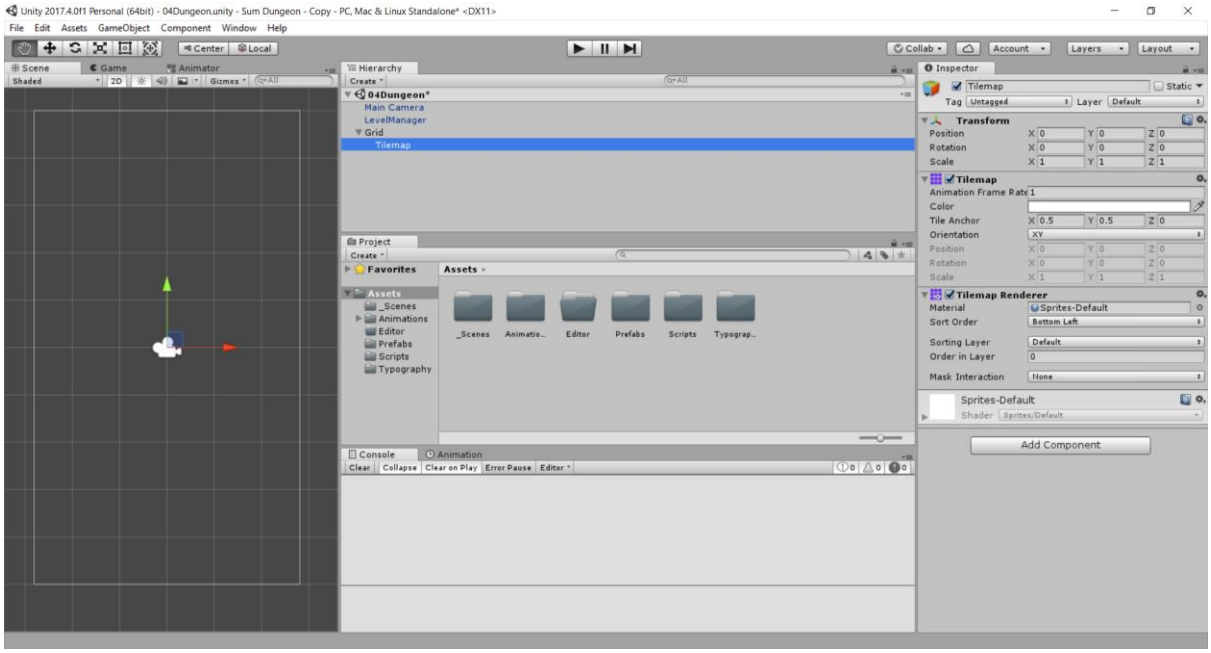
I have now met the success criteria 1.9, 2.4 and part of 1.4 which I specified for the New Game scene well as the validation for this scene I specified in the design section. The player can type the name of their character, select which operations they want to use and select a range of numbers they want to practice with. The validation will stop any inputs that will break the game from being accepted and the dialog box will tell players what they need to change. The player can go back to the Start Menu scene from this scene if they don't want to start a new game. I did a lot of testing in this section to make sure the validation worked and bug fixes to do with the dialog box. The test table helped me to keep track of all the tests I did and made sure I didn't miss any parts of the validation. I changed the code from my original design by mainly removing parts that were redundant and could be simplified. The validation I added also wasn't in my original pseudocode so adding it and the error messages changed the structure of the CreateNewGame class. The dialog box and the DialogErrorDisplay class was not part of my original plans but when programming the validation, I realised I needed some way of letting the user know if they entered invalid data and making sure my game didn't crash from invalid data. I decided to leave out generating the dungeon till the next scene since I wouldn't be able to test it without seeing the dungeon displayed on the Dungeon scene. Also, generating the dungeon map doesn't depend on any data the player entered on the New Game scene.

Scene 4: Dungeon

Creating the dungeon tile map

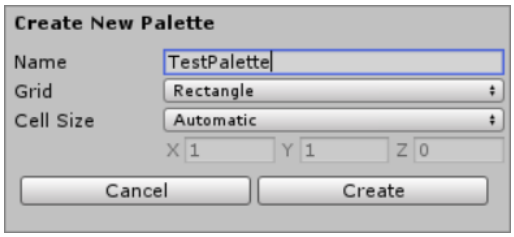
First, I added a tile map game object to the scene which will be used to store and display the dungeon (**figure 87, page 60**).

[Figure 87]



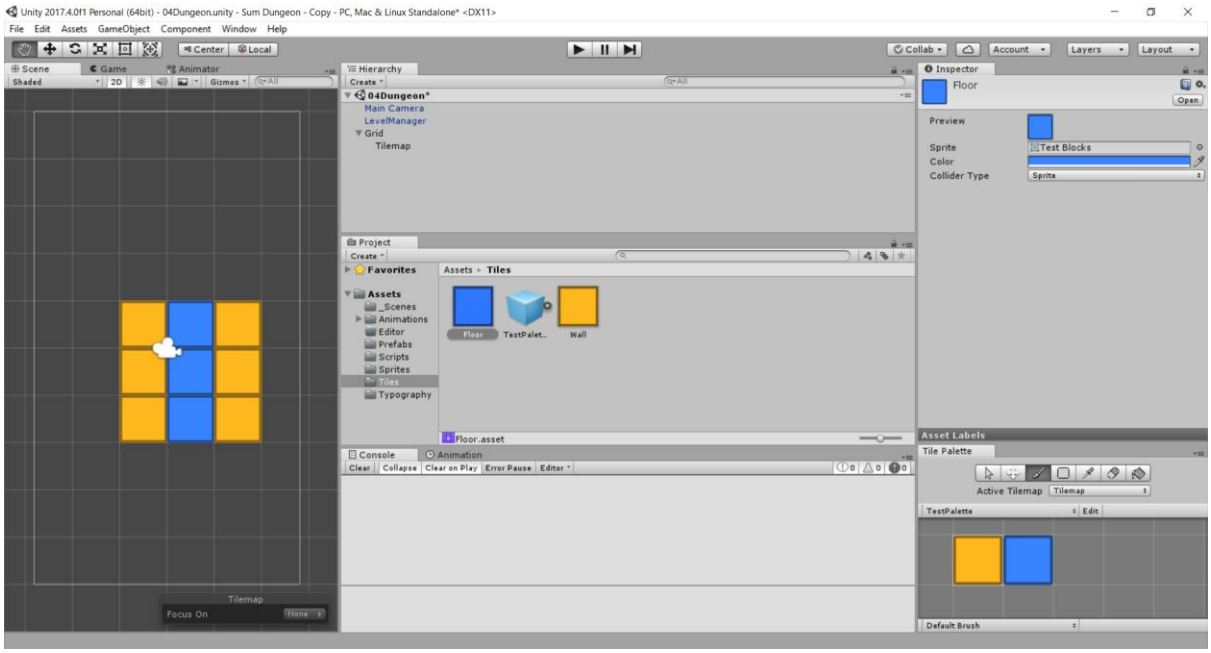
I then created a new tile palette for my tiles (**figure 88, page 60**). I created a test palette with simple tiles that I will eventually change but the test tiles are useful now for easily testing the dungeon.

[Figure 88]



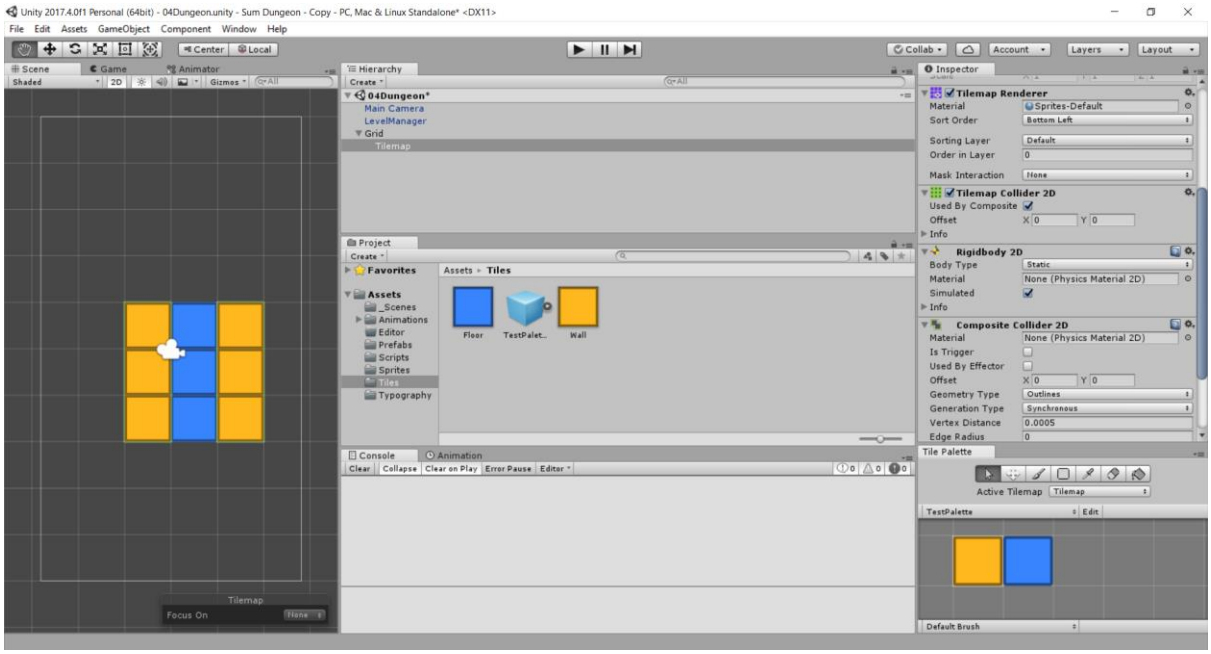
I added two basic tiles to the palette which were a blue floor and an orange wall. I checked they looked right by painting some tiles onto the scene (**figure 89, page 60**).

[Figure 89]



I changed the collider type of the floor to 'none' because I only want the walls to have colliders on them. I then added a tile map collider and a 2D composite collider to the tile map game object. The tile map collider adds colliders to each individual wall tile and the composite collider connects the individual tile colliders together so wall tiles that are adjacent form one collider which is more efficient than lots of small colliders. A 2D rigid body component was also added to the tile map but I set it to static, so it is not affected by gravity or other game physics. You can see the colliders on the tiles (The faint green outline around the groups of orange tiles) on the screenshot (**figure 91, page 61**).

[Figure 91]

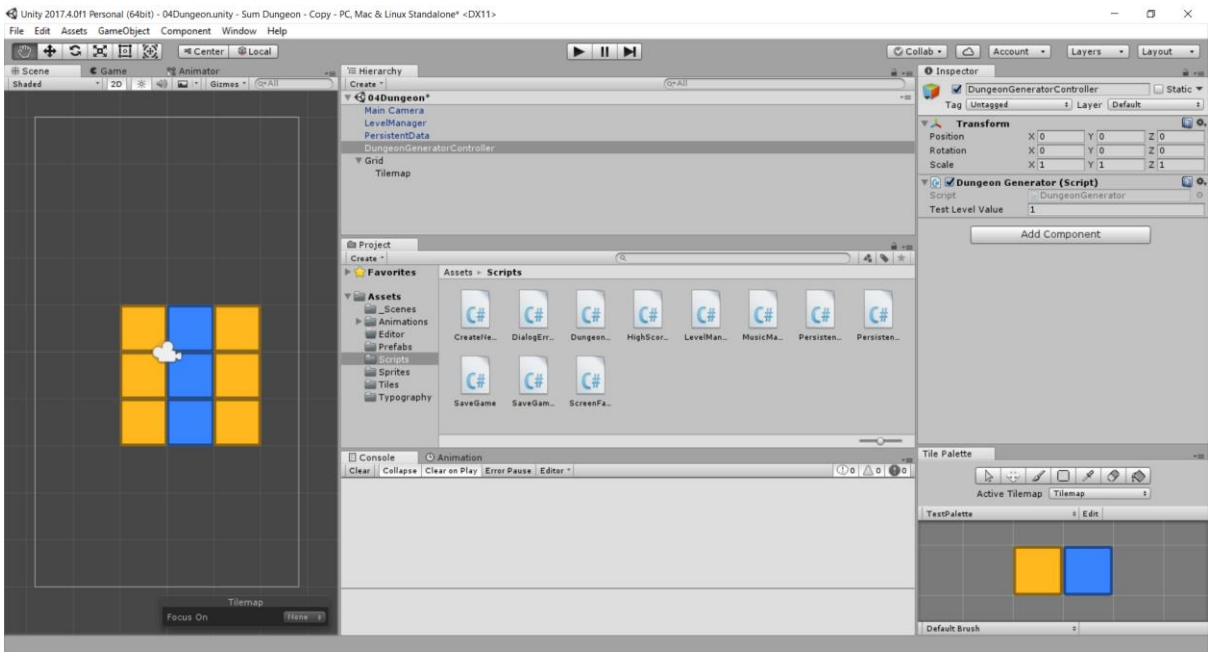


Developing the DungeonGenerator class

Task 1: Setting up the class and generating rooms

I then started to program the classes that would generate the dungeon. I first created the class that would become the template for the code representation of the tiles called CodeTile. I realised I could shorten the class by getting rid of the tile states attribute as the Closed tiles should always have a region number of 0 because they do not form any open and connected regions, so closed tiles would be tiles with a region number of 0 and open tiles will have a region number of any number except 0. That then made this class redundant as now all I needed to represent a tile was the region number, so I deleted this class and started making Dungeon Generator class just using a tile array of type integer which would be the region number. I first set up the DungeonGenerator class by giving it the 2D tile array of integers and access to the PersistentGameData class to read which dungeon level the player is on currently so that a harder dungeon can be created as the dungeon level increases. The dungeon map created by the DungeonGenerator also needs to be given to the PersistentGameData class when it has been generated to be able to save the dungeon. In Unity, I added the PersistentData game object to the Dungeon scene and I created a new game object for the DungeonGenerator class called Dungeon Generator Controller (**figure 92, page 62**). Since the DungeonGenerator class is so big, I will test as I program the different parts of it. I created a test level value that I could control the level of the dungeon with and see what the map would look like on different levels. I first created the methods to create a new dungeon grid and place rooms randomly in the grid both the grid and the rooms need to have odd sizes to work and I made the grid square for simplicity, but a rectangular grid could also work as long as the sides have odd lengths. While creating these two methods I also created a GetRandomOddNumber method to generate random odd numbers because I had to repeat code to do this. I refactored it into its own method to reduce the lines of code I had to write. I also created a method to help me debug the grid that would print a representation of the grid in Unity's console using the region numbers of each tile (**figure 93, page 63**).

[Figure 92]



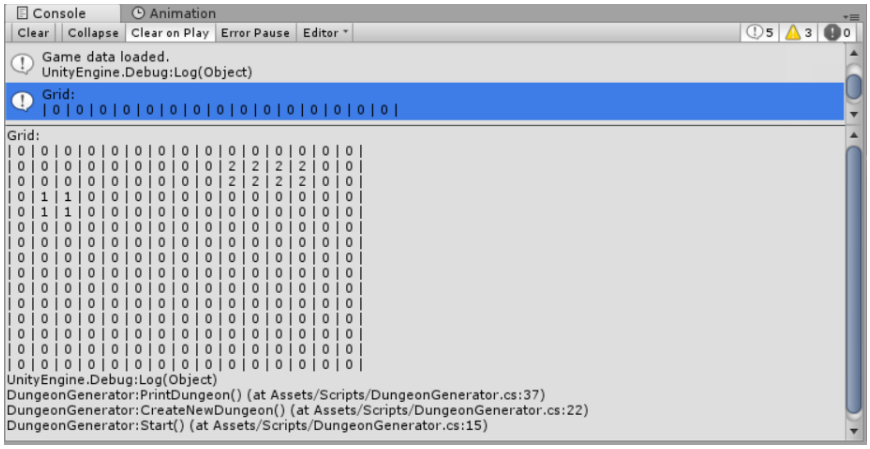
[Figure 93]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DungeonGenerator : MonoBehaviour {

    public int testLevelValue = 1;

    private int[,] tileArray;
    private PersistentGameData pgd;
    private int gridLength, newRegionNumber;

    void Start () {
        pgd = GameObject.FindObjectOfType<PersistentGameData>();
        CreateNewDungeon();
    }

    void CreateNewDungeon () {
        newRegionNumber = 1;
        CreateGrid(testLevelValue); //pgd.dungeonLevel will be used instead of testLevelValue when testing is done.
        PlaceRooms(testLevelValue); //pgd.dungeonLevel will be used instead of testLevelValue when testing is done.
        PrintDungeon();
    }
    //This method is just used for debugging to check the dungeon looks right.
    void PrintDungeon () {
        string printedGrid = "Grid:";
        for (int y = 0; y <= gridLength; y++) {
            string printedGridLine = "| ";
            for (int x = 0; x <= gridLength; x++) {
                printedGridLine = System.String.Concat(printedGridLine, tileArray[x,y].ToString(), " | ");
                if (x == gridLength) {
                    printedGrid = System.String.Concat(printedGrid, "\n", printedGridLine);
                }
            }
        }
        Debug.Log(printedGrid);
    }
    //Creates a square grid of wall tiles.
    void CreateGrid (int sideLength) {
        //The size of the grid increases as the player gets further, it must also always be an odd number.
        sideLength = (2 * sideLength) + 13;
        tileArray = new int[sideLength, sideLength];
        gridLength = sideLength - 1;
        for (int y = 0; y <= gridLength; y++) {
            for (int x = 0; x <= gridLength; x++) {
                tileArray[x,y] = 0;
            }
        }
    }

    //Attemps to place a number of rooms randomly on the grid, the greater the level of the dungeon the more
    //rooms it will try to place
    void PlaceRooms (int roomAttempts) {
        roomAttempts = roomAttempts + 2;
        int oddCoords = (gridLength / 2) - 2;
        for (int i = 1; i <= roomAttempts; i++) {
            //Find a random coordinate to place the room at.
            int xPos = GetRandomOddNumber(oddCoords);
            int yPos = GetRandomOddNumber(oddCoords);
            //Generate a random size for the room.
            int maxRoomSize = Mathf.RoundToInt(oddCoords/2);
            int xLength = GetRandomOddNumber(maxRoomSize);
            int yLength = GetRandomOddNumber(maxRoomSize);
            //Check the room won overlap with any other rooms and place the room on the grid.
            bool placeRoom = true;
```

```
68              for (int y = yPos; y <= (yPos + yLength); y++) {
69                  if (y > (gridLength-1)) {
70                      placeRoom = false;
71                      break;
72                  }
73                  for (int x = xPos; x <= (xPos + xLength); x++) {
74                      if (x > (gridLength-1) || tileArray[x,y] != 0) {
75                          placeRoom = false;
76                          break;
77                      }
78                  }
79              }
80              //Place the room if it didn't overlap.
81              if (placeRoom == true) {
82                  for (int y = yPos; y <= (yPos + yLength); y++) {
83                      for (int x = xPos; x <= (xPos + xLength); x++) {
84                          tileArray[x,y] = newRegionNumber;
85                      }
86                  }
87                  //Increase the number of the region so the next region has a different number.
88                  newRegionNumber++;
89              }
90          }
91      }
93      //Generates a random odd number between 1 and a maximum value.
94      int GetRandomOddNumber (int maxRange) {
95          int ranOddNum = Random.Range(0, maxRange+1);
96          ranOddNum = (2*ranOddNum) + 1;
97          return ranOddNum;
98      }
99  }
```

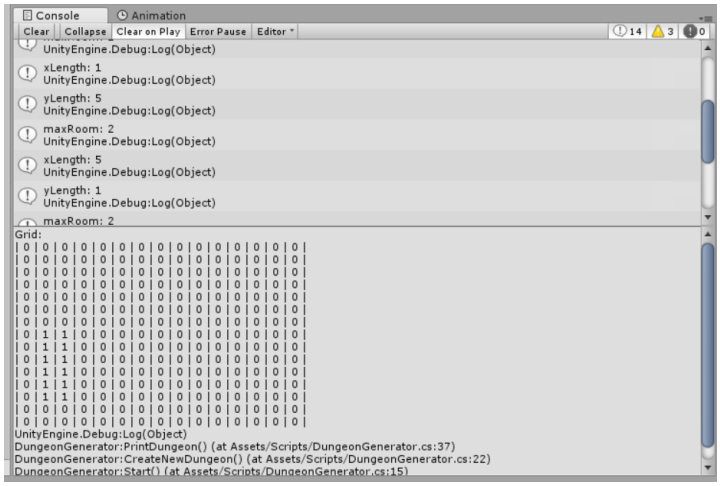
I then tested the current class to make sure it generated a grid and placed rooms that didn't overlap. It was successful as the rooms did not overlap however all the rooms had even sides (**figure 94, page 64**) which I didn't want because that messes up the rest of my program, so I had to fix it before progressing further.

[Figure 94]



I first started to print the maxRoomSize, xLength and yLength variables to see what was generated (**figure 95, page 65**). The maxRoom size was always the same as it is determined by the grid size which does not change so that was fine (the max size should be 5 tiles in either direction). The lengths of the rooms were also correct as they were all odd numbers between 1 and 5 so the problem should be with the loop that is changing the region number of the tiles.

[Figure 95]



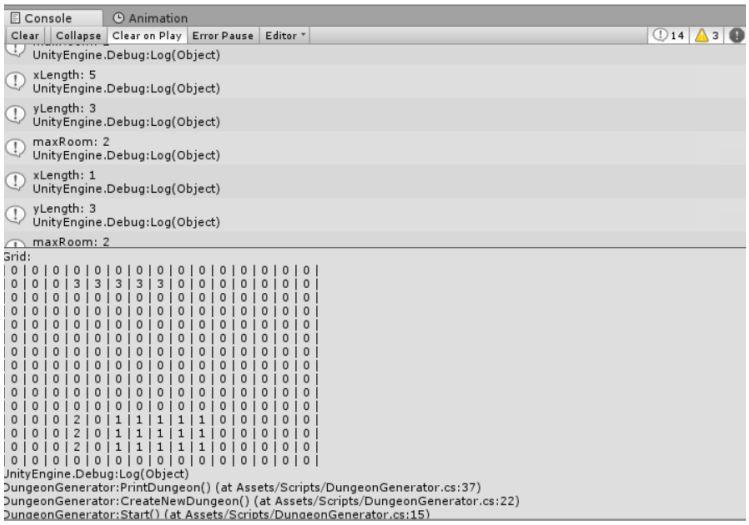
I realised I had made a logic error with the for loops because when they were looping through the areas where the rooms could be they were including the position of the origin point of the room + the length of the room when I didn't want them to include that point. I changed all the equal to or less than signs to just be less than signs (**figure 96, page 65**).

[Figure 96]

```
71              for (int y = yPos; y < (yPos + yLength); y++) {
72                  if (y > (gridLength-1)) {
73                      placeRoom = false;
74                      break;
75                  }
76                  for (int x = xPos; x < (xPos + xLength); x++) {
77                      if (x > (gridLength-1) || tileArray[x,y] != 0) {
78                          placeRoom = false;
79                          break;
80                      }
81                  }
82              }
83              //Place the room if it didn't overlap.
84              if (placeRoom == true) {
85                  for (int y = yPos; y < (yPos + yLength); y++) {
86                      for (int x = xPos; x < (xPos + xLength); x++) {
```

When I tested again the rooms now had all odd sides but some of the rooms had lengths or widths of 1 (**figure 97, page 65**) which I didn't want because only corridors can have the minimum length or width of one.

[Figure 97]



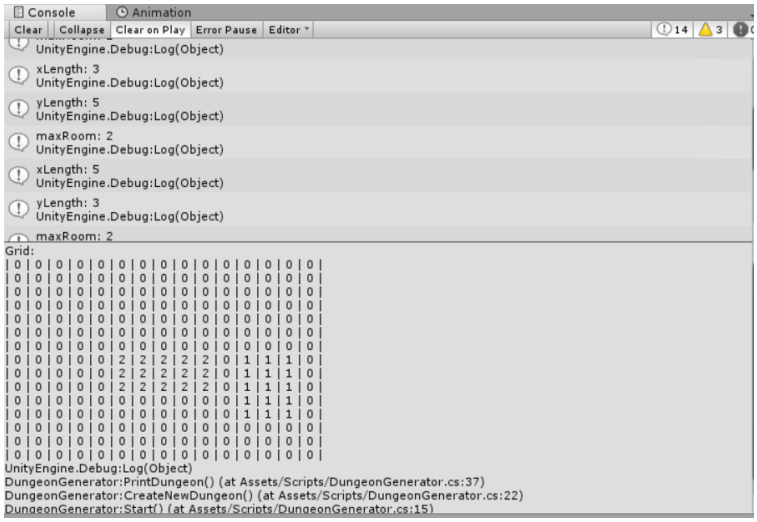
To get rid of this error I added some extra code to change the length and width of the rooms to 3 if they were 1 originally (**figure 98, page 66**).

[Figure 98]

```
65          int xLength = GetRandomOddNumber(maxRoomSize);
66          int yLength = GetRandomOddNumber(maxRoomSize);
67          if (xLength == 1) {
68              xLength = 3;
69          }
70          if (yLength == 1) {
71              yLength = 3;
72          }
```

I then tested the dungeon map again. This time all the errors were solved, and I was happy with the result (**figure 99, page 66**). I removed the previous debugging statements printing the lengths and widths of the rooms because I didn't need them anymore.

[Figure 99]



Task 2: Generating corridors

I could now move onto the next part of the generator: generating the corridors. This part of the program was very long so I refactored as much of the repeated code as I could to split up the GenerateCorridors method, so it wasn't as long and easier to read. I added an extra three methods to the program from this refactoring called AreAllOddTilesOpen, FindFrontierTiles and GetCardinalTileCoords (**figure 100, page 67**). The GenerateCorridors method runs a maze generator through all the spaces around the rooms and stops when all the odd tiles have been opened.

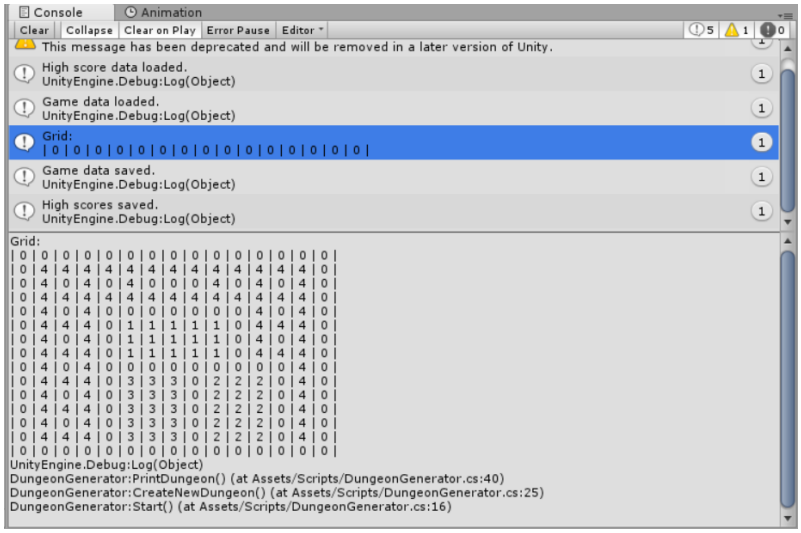
[Figure 100]

```
109    //Generates corridors that fill the spaces between the rooms using a maze generator based on Prim's algorithm.
110    void GenerateCorridors () {
111        //Find a closed tile to start the maze generator at.
112        Vector2Int startPoint = AreAllOddTilesOpen();
113        //Check all the odd tiles are open to finish running the maze generator.
114        do { //Maze generator.
115            //Open the tile with the starting tile coordinates.
116            tileArray[startPoint.x,startPoint.y] = newRegionNumber;
117            //Find frontier tiles.
118            FindFrontierTiles(startPoint.x, startPoint.y);
119            //While the list of frontier cells is not empty
120            while (frontierTilesList.Count != 0) {
121                //Pick a random frontier tile from the list of frontier tiles.
122                int randomFrontier = Random.Range(0, frontierTilesList.Count);
123                Vector2Int frontierTile = frontierTilesList[randomFrontier];
124                //Find the neighbors of the fontier cell
125                Vector2Int[] neighbours = GetCardinalTileCoords(frontierTile.x, frontierTile.y);
126                List<Vector2Int> connectNeighbours = new List<Vector2Int>();
127                for (int i = 0; i <= 3; i++) {
128                    if (neighbours[i] != Vector2Int.zero && tileArray[neighbours[i].x, neighbours[i].y] == newRegionNumber)
129                        connectNeighbours.Add(neighbours[i]);
130                    }
131                }
132                //Pick a random neighbour and connect it to the frontier tile.
133                int randomNeighbour = Random.Range(0, connectNeighbours.Count);
134                Vector2Int pickedNeighbour = connectNeighbours[randomNeighbour];
135                tileArray[frontierTile.x, frontierTile.y] = newRegionNumber;
136                    tileArray[pickedNeighbour.x, pickedNeighbour.y] = newRegionNumber;
137                    int connectingX = (frontierTile.x + pickedNeighbour.x) / 2;
138                    int connectingY = (frontierTile.y + pickedNeighbour.y) / 2;
139                    tileArray[connectingX, connectingY] = newRegionNumber;
140                    //Remove the frontier tile from the list of frontier tile.
141                    frontierTilesList.Remove(frontierTile);
142                    //Find the frontier tiles of the current frontier tile.
143                    FindFrontierTiles(frontierTile.x, frontierTile.y);
144                }
145                newRegionNumber++;
146                startPoint = AreAllOddTilesOpen();
147            } while (startPoint != Vector2Int.zero);
148        }
150        //Loops through all tiles with odd coordinates and returns the coordinate of the first tile that is still closed,
151        //Otherwise it returns the coordinates of the origin.
152        Vector2Int AreAllOddTilesOpen () {
153            for (int y = 1; y < gridLength; y += 2) {
154                for (int x = 1; x < gridLength; x += 2) {
155                    if (tileArray[x,y] == 0) {
156                        Vector2Int emptyTile = new Vector2Int(x, y);
157                        return emptyTile;
158                    }
159                }
160            }
161            return Vector2Int.zero;
162        }
163
164        //Adds the frontier tiles around a tile with the coordinates given to the frontier tiles list.
165        void FindFrontierTiles (int xCoord, int yCoord) {
166            Vector2Int[] NESWtiles = GetCardinalTileCoords(xCoord, yCoord);
167            for (int i = 0; i <= 3; i++) {
168                if (NESWtiles[i] != Vector2Int.zero && tileArray[NESWtiles[i].x, NESWtiles[i].y] == 0) {
169                    frontierTilesList.Add(NESWtiles[i]);
170                }
171            }
172        }
```

```
174     //Get the coordinates of each tile two away from the origional coordinate in every cardinal direction.
175     Vector2Int[] GetCardinalTileCoords (int xCoord, int yCoord) {
176         Vector2Int[] NESWArray = new Vector2Int[4];
177         NESWArray [0] = new Vector2Int (xCoord, yCoord + 2);
178         NESWArray [1] = new Vector2Int (xCoord + 2, yCoord);
179         NESWArray [2] = new Vector2Int (xCoord, yCoord - 2);
180         NESWArray [3] = new Vector2Int (xCoord - 2, yCoord);
181         //Get rid of coordinates that are outside the grid.
182         for (int i = 0; i <= 3; i++) {
183             if (NESWArray [i].x > gridLength || NESWArray [i].x < 0) {
184                 NESWArray [i] = Vector2Int.zero;
185             } else if (NESWArray [i].y > gridLength || NESWArray [i].y < 0) {
186                 NESWArray[i] = Vector2Int.zero;
187             }
188         }
189         return NESWArray;
190     }
191 }
```

I then tested the class again to check it was creating a maze of corridors around the rooms. It was successful in the way that it generated corridors around the rooms but there was a problem with the maze generator because it would loop in on itself where there should currently be dead ends which I didn't want to happen (**figure 101, page 68**). The maze generator should create a minimum spanning tree between all the tiles with odd coordinates.

[Figure 101]



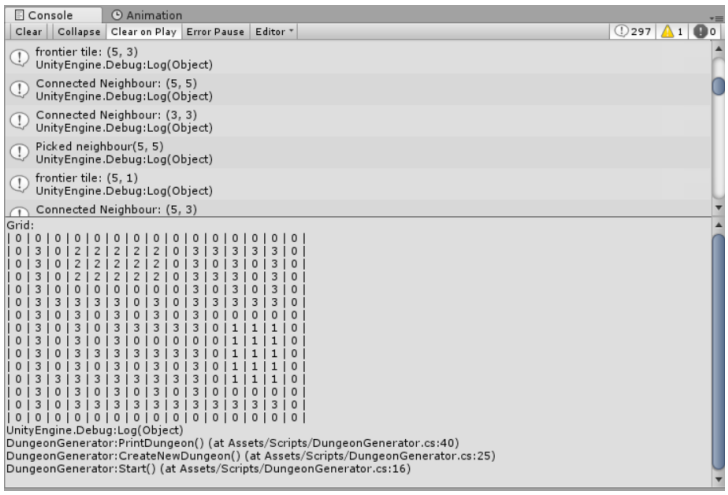
I thought the problem might be with the neighbours a tile finds when looking for tiles to connect to, so I decided to print the variables frontierTile, all connectedNeighbours and the pickedNeighbour. After this I noticed that the grid was being printed upside-down, so I edited the for loop that iterates through the y coordinates of the PrintDungeon method to print it the right way around, so I could read the coordinates that I was printing properly (**figure 102, page 68**).

[Figure 102]

```
28      //This method is just used for debugging to check the dungeon looks right.
29      void PrintDungeon () {
30          string printedGrid = "Grid:";
31          for (int y = gridLength; y >= 0; y--) {
32              string printedGridLine = "| ";
33              for (int x = 0; x <= gridLength; x++) {
34                  printedGridLine = System.String.Concat(printedGridLine, tileArray[x,y].ToString(), " | ");
35                  if (x == gridLength) {
36                      printedGrid = System.String.Concat(printedGrid, "\n", printedGridLine);
37                  }
38              }
39          }
40          Debug.Log(printedGrid);
41      }
```

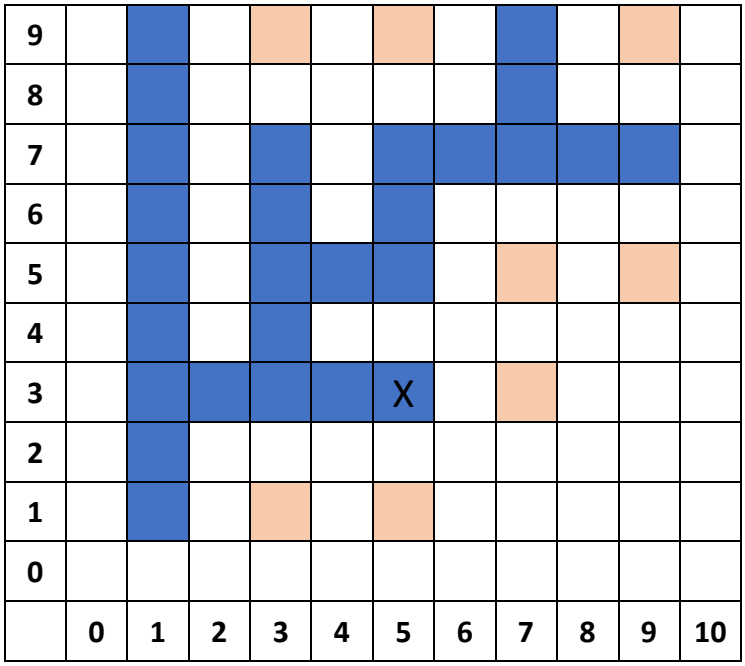
I tested again to view the variables that I was printing (**figure 103, page 69**).

[Figure 103]



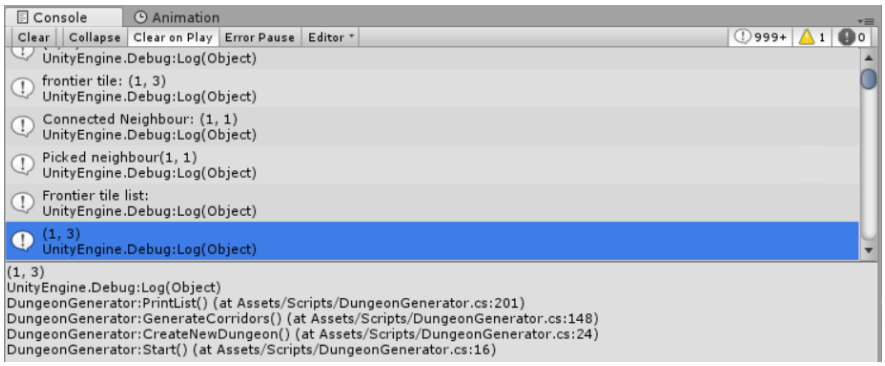
I then did a dry run of the algorithm myself using the variables being printed to draw my own part of the dungeon to see where the algorithm was going wrong (**figure 104, page 69**) and this is what I found; after generating the path properly after a few iterations, the program went back to a tile that had already been opened (marked with an X on my diagram) and should not have been in the list of frontier tiles. For some reason, the open tiles should have been removed from the frontier list but were not so the program would go back and treated open tiles as ones that had not already been connected when they were actually connected leading to multiple connections which I didn't want.

[Figure 104]



Now I knew what the problem was I looked at what was in the frontier tiles list after each iteration to see if coordinates were being removed. I found they weren't being removed (**figure 105, page 70**), which was causing a problem.

[Figure 105]



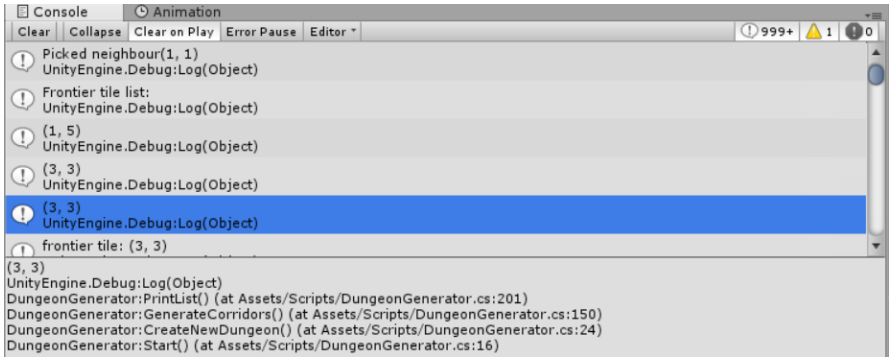
Even though there is a line in my code that should remove the coordinate it did not seem to be working so I tried a different way of removing it (**figure 106, page 70**). Since I knew the index of the coordinate I used that to reference where the coordinate I wanted to remove was located at.

[Figure 106]

```
145             //Remove the frontier tile from the list of frontier tile.
146             frontierTilesList.RemoveAt(randomFrontier);
147             //Find the frontier tiles of the current frontier tile.
148             FindFrontierTiles(frontierTile.x, frontierTile.y);
149             Debug.Log("Frontier tile list: ");
150             PrintList();
```

I tested my program again and although the coordinates were now removed properly, duplicates of some of the coordinates appeared (**figure 107, page 70**) which would also cause problems.

[Figure 107]



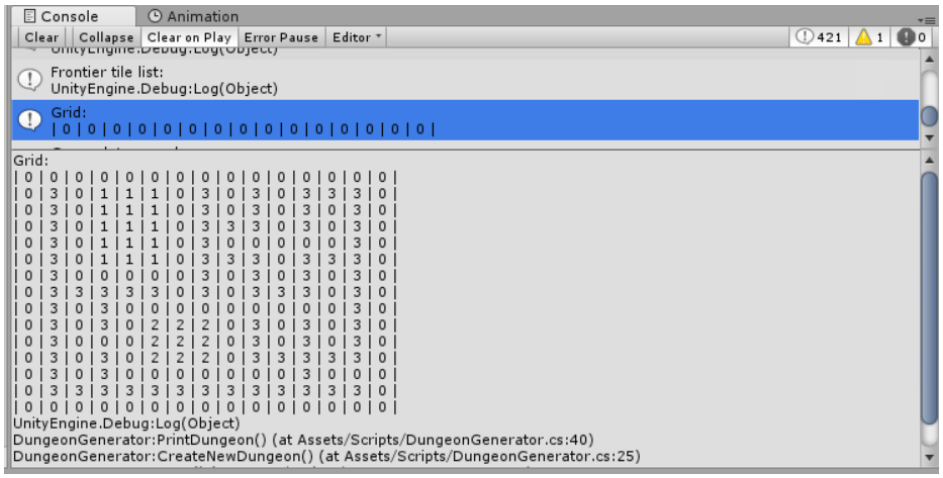
After trying the test again, the same coordinates were repeated which I realised was because when my program was finding new frontier tiles, it found tiles that were already in the frontierTileList. To fix this, I added a condition when finding new frontier tiles that meant they could only be added to the list if they weren't already in it (**figure 108, page 70**).

[Figure 108]

```
172     //Adds the frontier tiles around a tile with the coordinates given to the frontier tiles list if it is not
173     //already in the list.
174     void FindFrontierTiles (int xCoord, int yCoord) {
175         Vector2Int[] NESWtiles = GetCardinalTileCoords(xCoord, yCoord);
176         for (int i = 0; i <= 3; i++) {
177             if (NESWtiles[i] != Vector2Int.zero && tileArray[NESWtiles[i].x, NESWtiles[i].y] == 0) {
178                 if (frontierTilesList.Contains(NESWtiles[i]) == false) {
179                     frontierTilesList.Add(NESWtiles[i]);
180                 }
181             }
182         }
183     }
```

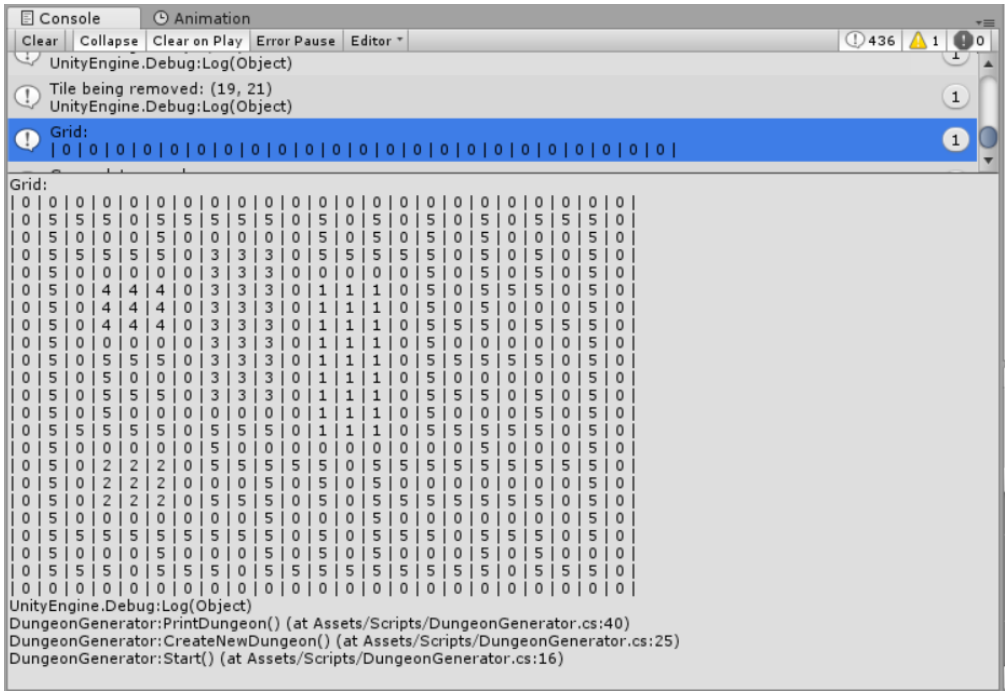
I tested the class again and this time the maze corridors were created properly (**figure 109, page 71**).

[Figure 109]



Since the grid I was testing with was small, the corridors tended to be quite straight as there wasn't a lot of space to place the corridors, so I tried generating a dungeon that would be made from a later dungeon level of 5. It also worked, and this dungeon was generated on a larger grid, so it was created with 4 rooms with the corridors being the 5th region (**figure 110, page 71**).

[Figure 110]



I then removed all the debugging code I added that displayed the different variables because I didn't need it any more.

Task 3: Connecting the corridors to the rooms

At the moment none of the areas in my dungeons connect to each other so the next task was to unify the dungeons. The first part is to find all the tiles that could connect adjacent regions. I created a new method to create a list of the coordinates of these tiles and the two regions they connect (**figure 111, page 72**). It works by iterating through all the tiles and checking if it could connect two

tiles that have different region numbers. Rooms can be connected directly to each other using this method because each room has a different region number. The second part is to pick a random connector around region 1 and connect it to another region, then give the new connected region a region number of 1, repeat this to connect all the regions until they form one region with a region number of 1. As two regions become connected, remove the connecting tiles that connect them from the connectors list and open up some extra connecting tiles so that there isn't just one path through the dungeon.

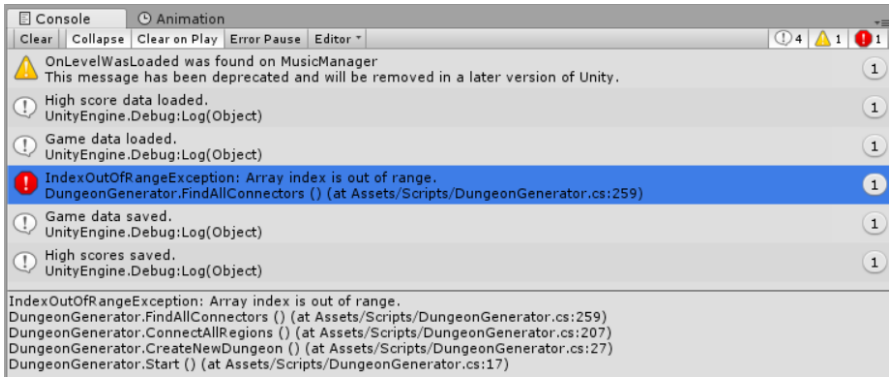
[Figure 111]

```
205     //Connect all rooms and corridors together to form one region.
206     void ConnectAllRegions () {
207         FindAllConnectors();
208         int connectingRegion = 0;
209         //Connect the regions starting from regions 1 and 2.
210         for (int i = 2; i < newRegionNumber; i++) {
211             //Find a connector that connects to region 1.
212             bool connectsTo1 = false;
213             int[] randomConnector;
214             do {
215                 randomConnector = connectors[Random.Range(0, connectors.Count)];
216                 if (randomConnector[2] == 1) {
217                     connectingRegion = randomConnector[3];
218                     connectsTo1 = true;
219                 } else if (randomConnector[3] == 1) {
220                     connectingRegion = randomConnector[2];
221                     connectsTo1 = true;
222                 }
223             } while (connectsTo1 == false);
224             //Open the connector.
225             tileArray[randomConnector[0],randomConnector[1]] = 1;
226             //Convert the region number of the newly connected region to 1.
227             for (int y = 1; y < gridLength; y++) {
228                 for (int x = 1; x < gridLength; x++) {
229                     if (tileArray[x,y] == connectingRegion) {
230                         tileArray[x,y] = 1;
231                     }
232                 }
233             }
234             //Remove all extraneous connectors between the two regions that are now connected but give them
235             //some chance of opening up.
236             for (int j = 0; j < connectors.Count; j++) {
237                 if (connectors[j][2] == 1 && connectors[j][3] == connectingRegion) {
238                     OpenUpChance(connectors[j][0], connectors[j][1]);
239                     connectors.Remove(connectors[j]);
240                 } else if (connectors[j][2] == connectingRegion && connectors[j][3] == 1) {
241                     OpenUpChance(connectors[j][0], connectors[j][1]);
242                     connectors.Remove(connectors[j]);
243                 }
244             }
245         }
246     }
```

```
248     //Finds tiles that could connect two different regions.
249     void FindAllConnectors () {
250         //Iterate through all tiles.
251         for (int y = 1; y < gridLength; y++) {
252             for (int x = 1; x < gridLength; x++) {
253                 //Only check blocked tiles.
254                 if (tileArray[x,y] == 0) {
255                     //Check the tiles north and south for different regions.
256                     if (tileArray[x,y+2] != 0 && tileArray[x,y+2] != tileArray[x,y-2]) {
257                         connectors.Add(new int[4] {x, y, tileArray[x,y+2], tileArray[x,y-2]});
258                     //Check the tiles east and west for different regions.
259                     } else if (tileArray[x+2,y] != 0 && tileArray[x+2,y] != tileArray[x-2,y]) {
260                         connectors.Add(new int[4] {x, y, tileArray[x+2,y], tileArray[x-2,y]});
261                     }
262                 }
263             }
264         }
265     }
266
267     //Give a tile a 1 in 12 chance of opening up.
268     void OpenUpChance (int xCoords, int yCoords) {
269         int chance = Random.Range(1, 13);
270         if (chance == 1) {
271             tileArray[xCoords, yCoords] = 1;
272         }
273     }
274
275 }
```

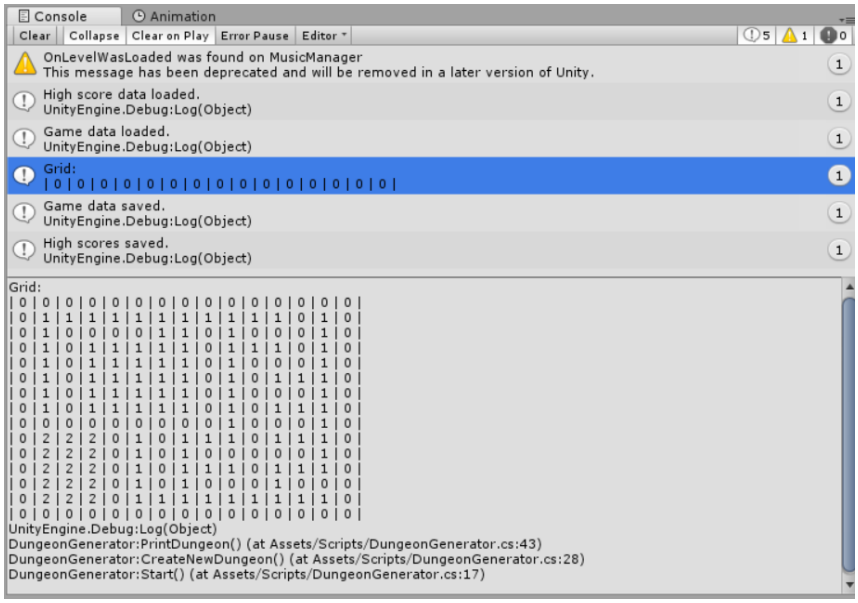
I now tested the DungeonGenerator class to make sure it connected all the different numbered regions and now all open regions have a region number of 1. The dungeon was not generated because I got an error from an array index being out of range (**figure 112, page 73**).

[Figure 112]

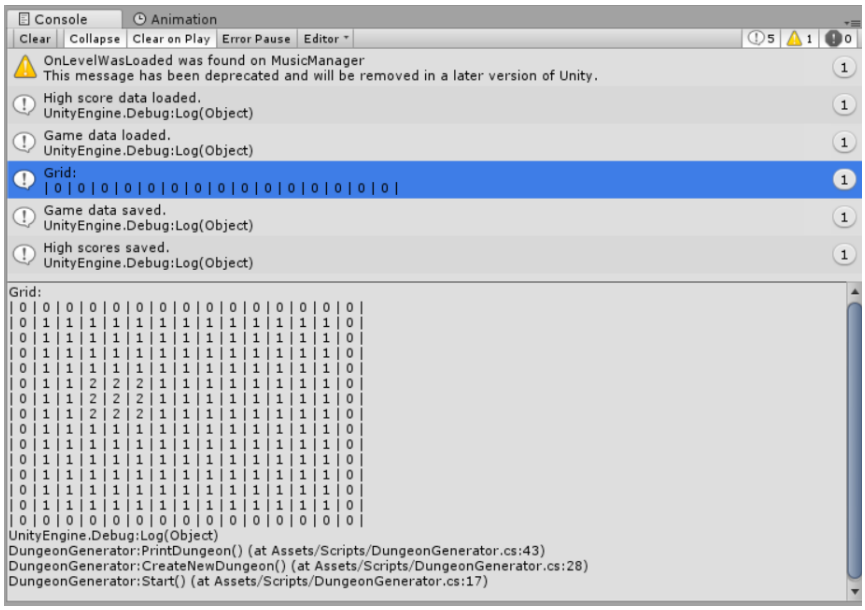


I realised this was because when my program was looking for connecting tiles, I was checking the tiles 2 tiles away from the connecting tile instead of 1. I made changes to the FindAllConnectors method to stop this from happening. I tested again and this time, there was no error and a grid displayed. However, my program wasn't working properly yet as only two regions were connected, and one room wasn't (**figure 113, page 74**). When I did the test a few times again it seemed like, whenever more than 3 regions were generated, the whole board would end up connected which got rid of the whole dungeon (**figure 114, page 74**). Sometimes my game would also freeze, crashing Unity.

[Figure 113]



[Figure 114]



To try and stop my game crashing I got rid of the do-while loop and changed my code so that instead of looking for the first region created, it would start with any region and connect all other regions to itself and give them its region number (**figure 115, page 75**). I tried testing again and now my game no longer crashed so I think the crashing was caused by the do-while loop I got rid of. I also edited the OpenUpChance method to give connectors a higher chance of opening up and there being multiple paths (**figure 116, page 75**).

[Figure 115]

```
205     //Connect all rooms and corridors together to form one region.
206     void ConnectAllRegions () {
207         FindAllConnectors();
208         //Connect the regions starting from regions 1 and 2.
209         for (int i = 2; i < newRegionNumber; i++) {
210             //Get a random connector tile and connect the two regions it connects.
211             int[] randomConnector = connectors[Random.Range(0, connectors.Count)];
212             int startingRegion = randomConnector[2];
213             int connectingRegion = randomConnector[3];
214             //Open the connector.
215             tileArray[randomConnector[0],randomConnector[1]] = startingRegion;
216             //Convert the region number of the newly connected region to the number of the starting region.
217             for (int y = 1; y < gridLength; y++) {
218                 for (int x = 1; x < gridLength; x++) {
219                     if (tileArray[x,y] == connectingRegion) {
220                         tileArray[x,y] = startingRegion;
221                     }
222                 }
223             }
224             //Remove all extraneous connectors between the two regions that are now connected but give them
225             //some chance of opening up.
226             for (int j = 0; j < connectors.Count; j++) {
227                 if (connectors[j][2] == startingRegion && connectors[j][3] == connectingRegion) {
228                     OpenUpChance(con                    ors[j][1], connectingRegion);
229                     connectors.Remov
230                 } else if (connectors[j][2] == connectingRegion && connectors[j][3] == startingRegion) {
231                     OpenUpChance(connectors[j][0], connectors[j][1], connectingRegion);
232                     connectors.Remove(connectors[j]);
233                 }
234             }
235         }
236     }
```

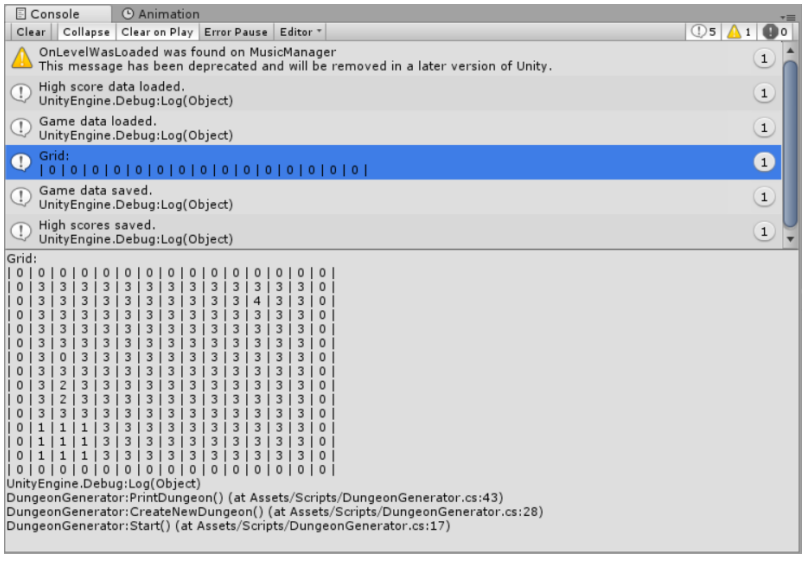
int startingRegion

[Figure 116]

```
257     //Give a tile a 1 in 7 chance of opening up.
258     void OpenUpChance (int xCoords, int yCoords, int connectToRegion) {
259         int chance = Random.Range(1, 8);
260         if (chance == 1) {
261             tileArray[xCoords, yCoords] = connectToRegion;
262         }
263     }
```

The other errors were still present when I tested just now, the connected parts of my dungeon didn't always have a region number of 1 (**figure 117, page 75**).

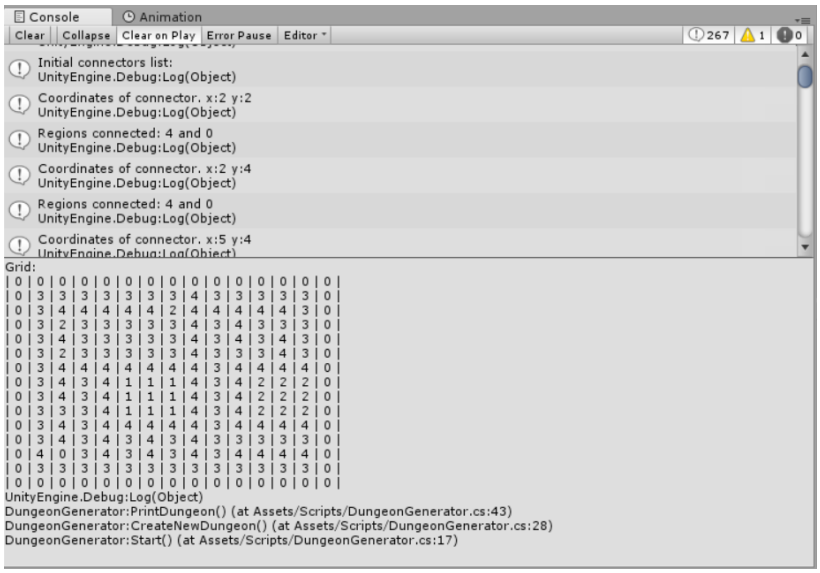
[Figure 117]



To find out what was going wrong I decided to print the values stored by the connectors list.

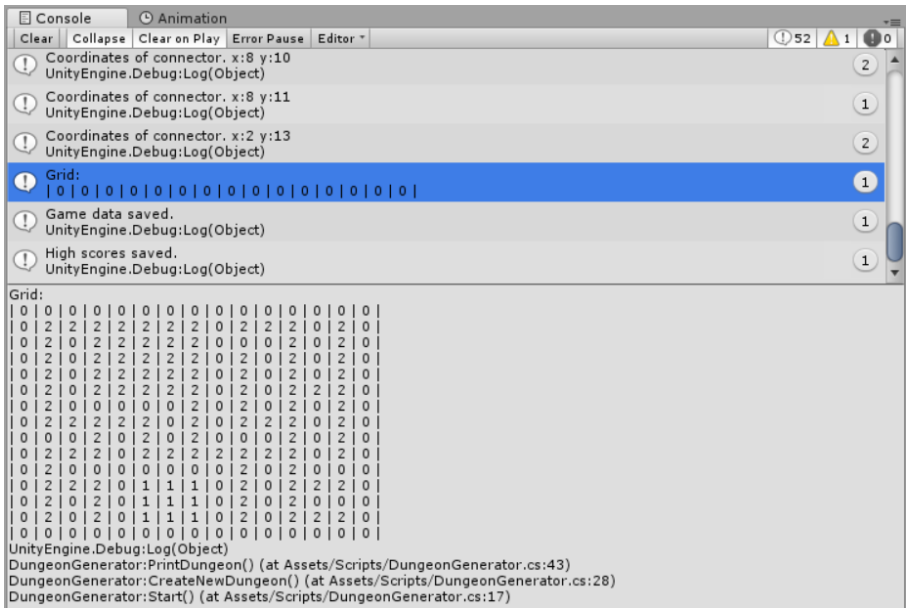
When I tested again and the values in the connectors list showed connecting tiles were being found between closed and open tiles, regions with a region number of zero and not zero respectively (**figure 118, page 76**). This is what was causing the grid to fill with open regions.

[Figure 118]



I changed my code to fix this by giving the FindAllConnectors method stricter conditions for finding a connecting tile. When I tested my code again, the grid no longer filled up with open tiles, but one area would sometimes still not be connected (**figure 119, page 76**).

[Figure 119]



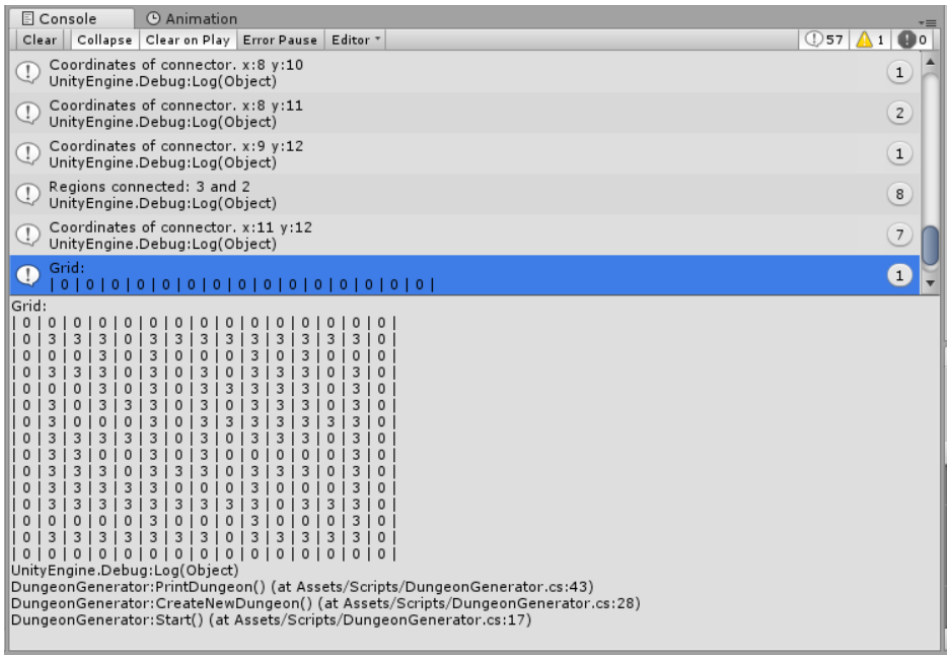
To fix this I changed the condition to finish running the ConnectAllRooms method from a for loop to checking when the list of connecting tiles was empty. I also spotted and fixed an error with the OpenUpChance method that was giving the extra connected tiles the wrong region number and changed the chance of a connector opening up to make it smaller (**figure 120, page 77**).

[Figure 120]

```
198     //Connect all rooms and corridors together to form one region.
199     void ConnectAllRegions () {
200         FindAllConnectors();
201         Debug.Log("Initial connectors list: ");
202         PrintList();
203         //Connect the regions until the list of connecting tiles is empty.
204         while (connectors.Count != 0) {
205             //Get a random connector tile and connect the two regions it connects.
206             int[] randomConnector = connectors[Random.Range(0, connectors.Count)];
207             int startingRegion = randomConnector[2];
208             int connectingRegion = randomConnector[3];
209             //Open the connector.
210             tileArray[randomConnector[0],randomConnector[1]] = startingRegion;
211             //Convert the region number of the newly connected region to the number of the starting region.
212             for (int y = 1; y < gridLength; y++) {
213                 for (int x = 1; x < gridLength; x++) {
214                     if (tileArray[x,y] == connectingRegion) {
215                         tileArray[x,y] = startingRegion;
216                     }
217                 }
218             }
219             //Remove all extraneous connectors between the two regions that are now connected but give them
220             //some chance of opening up.
221             for (int j = 0; j < connectors.Count; j++) {
222                 if (connectors[j][2] == startingRegion && connectors[j][3] == connectingRegion) {
223                     OpenUpChance(connectors[j][0], connectors[j][1], startingRegion);
224                     connectors.Remove(connectors[j]);
225                 } else if (connectors[j][2] == connectingRegion && connectors[j][3] == startingRegion) {
226                     OpenUpChance(connectors[j][0], connectors[j][1], startingRegion);
227                     connectors.Remove(connectors[j]);
228                 }
229             }
230             Debug.Log("Connectors list after each iteration: ");
231             PrintList();
232         }
233     }
235     //Finds tiles that could connect two different regions.
236     void FindAllConnectors () {
237         //Iterate through all tiles.
238         for (int y = 1; y < gridLength; y++) {
239             for (int x = 1; x < gridLength; x++) {
240                 //Only check blocked tiles.
241                 if (tileArray[x,y] == 0) {
242                     //Check the tiles north and south for different regions.
243                     if (tileArray[x,y+1] != 0 && tileArray[x,y-1] != 0 && tileArray[x,y+1] != tileArray[x,y-1]) {
244                         connectors.Add(new int[4] {x, y, tileArray[x,y+1], tileArray[x,y-1]});
245                     //Check the tiles east and west for different regions.
246                     } else if (tileArray[x+1,y] != 0 && tileArray[x-1,y] != 0 && tileArray[x+1,y] != tileArray[x-1,y]) {
247                         connectors.Add(new int[4] {x, y, tileArray[x+1,y], tileArray[x-1,y]});
248                     }
249                 }
250             }
251         }
252     }
253
254     //Give a tile a 1 in 10 chance of opening up.
255     void OpenUpChance (int xCoords, int yCoords, int connectToRegion) {
256         int chance = Random.Range(1, 11);
257         if (chance == 1) {
258             tileArray[xCoords, yCoords] = connectToRegion;
259         }
260     }
```

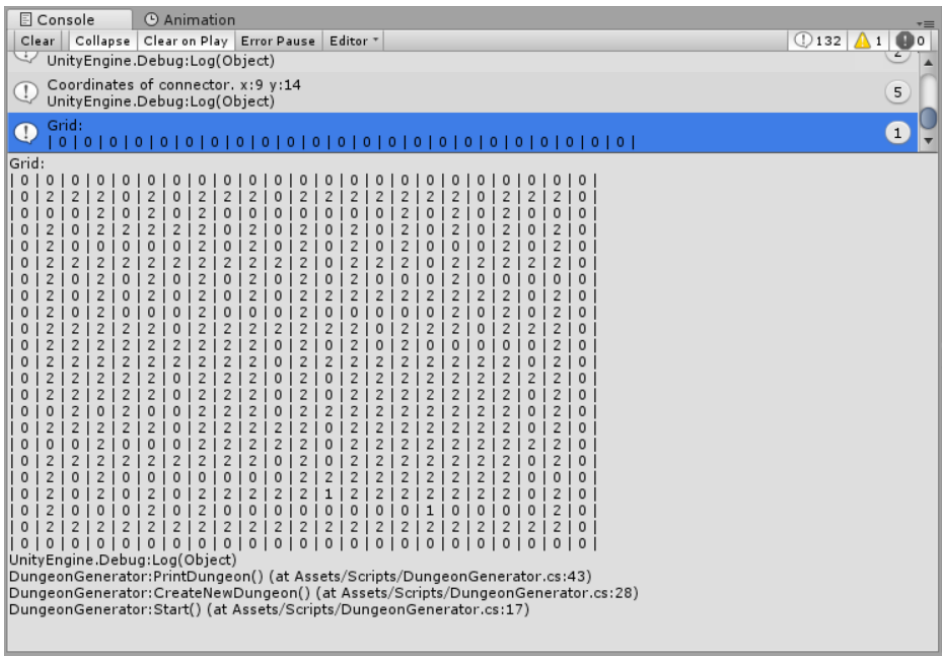
When I tested this time, the test was successful and all the open regions in the dungeon were connected together (**figure 121, page 78**).

[Figure 121]



I then tested with a test dungeon level value of 5 to check the dungeon was being generated properly when it was bigger (**figure 122, page 78**). It worked fine but sometimes there would be a few tiles that didn't have the same region number as the rest. These were probably caused by the OpenUpChance method adding extra connectors but since they didn't affect the dungeon in any detrimental way, I decided to leave them.

[Figure 122]



I removed the debugging code from my methods before continuing to the next task.

Task 4: Removing dead ends

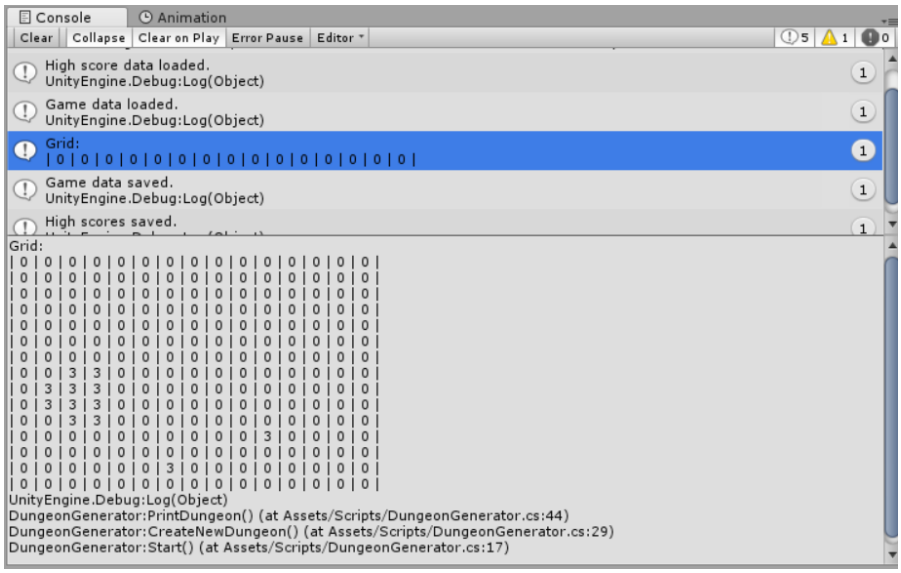
This step isn't necessary for the dungeon to be traversable, but it will lessen player frustration from repeatably running into dead ends and make sure all paths lead somewhere. Removing all the dead ends is done by removing all open tiles that have walls on 3 sides until there are no more left. I did this by recursively calling the method to remove dead ends because when one dead end tile is closed it creates another dead-end tile until the whole dead-end path has been closed (**figure 123, page 79**).

[Figure 123]

```
267     //The method that is called from the main CreateNewDungeon method and starts off the recursive
268     //removal of dead ends.
269     void RemoveDeadEnds () {
270         Vector2Int deadEnd = FindDeadEnds();
271         RecursiveRemove(deadEnd);
272     }
273 
274     //Recursivley call this method until there are no more deadEnds.
275     void RecursiveRemove (Vector2Int currentTile) {
276         //Close the dead end.
277         tileArray[currentTile.x, currentTile.y] = 0;
278         Vector2Int nextDeadEnd = CountWalls(currentTile.x, currentTile.y);
279         if (nextDeadEnd != Vector2Int.zero) {
280             RecursiveRemove(nextDeadEnd);
281         } else {
282             Vector2Int newDeadEnd = FindDeadEnds();
283             if (newDeadEnd != Vector2Int.zero) {
284                 RecursiveRemove(newDeadEnd);
285             }
286         }
287     }
289     //Find a new dead end and return its coordinates, otherwise return (0,0).
290     Vector2Int FindDeadEnds () {
291         for (int y = gridLength-1; y >= 1; y--) {
292             for (int x = 1; x < gridLength; x++) {
293                 if (tileArray[x,y] != 0) {
294                     if (CountWalls(x,y) != Vector2Int.zero) {
295                         Vector2Int newDeadEnd = new Vector2Int(x,y);
296                         return newDeadEnd;
297                     }
298                 }
299             }
300         }
301         return Vector2Int.zero;
302     }
304     //If the given tile is a dead end, return the coordinates of the tile connecting it to the
305     //rest of the dungeon, otherwise return the coordinates (0,0).
306     Vector2Int CountWalls (int xCoord, int yCoord) {
307         int wallCount = 0;
308         Vector2Int openSide = new Vector2Int(0,0);
309         Vector2Int[] NESWArray = new Vector2Int[4];
310         NESWArray [0] = new Vector2Int (xCoord, yCoord + 1);
311         NESWArray [1] = new Vector2Int (xCoord + 1, yCoord);
312         NESWArray [2] = new Vector2Int (xCoord, yCoord - 1);
313         NESWArray [3] = new Vector2Int (xCoord - 1, yCoord);
314         foreach (Vector2Int v in NESWArray) {
315             if (tileArray[v.x,v.y] == 0) {
316                 wallCount++;
317             } else {
318                 openSide = v;
319             }
320         }
321         if (wallCount == 3) {
322             return openSide;
323         } else {
324             return Vector2Int.zero;
325         }
326     }
```

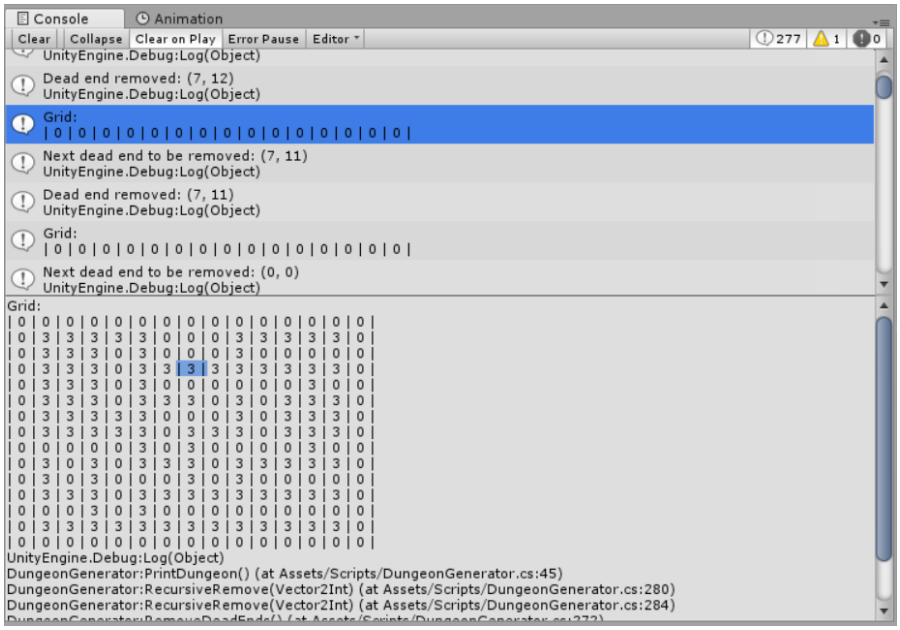
I then started testing the last part of the dungeon generator to test if it now created dungeons with rooms and corridors that all connect and without any dead ends. The first test was unsuccessful as almost all the dungeon got removed by the removal of dead ends (**figure 124, page 80**).

[Figure 124]



To see what was going wrong I printed all the coordinates of the dead-end tiles to be removed and the grid after a tile was removed (**figure 125, page 80**). I found out that my program was removing tiles that weren't dead ends for some reason.

[Figure 125]



I decided to trace through my program to figure out where my program wasn't working. I used the data of my previous test to trace through. The part of the dungeon I traced through showed that when finding the next tile to remove, my program was finding it from the tile that was just removed and not checking whether that tile was also a dead end (**figure 126, page 81**).

[Figure 126]

13	3				3
12	3		X		3
11	3	3	3	3	3
10	3				
9	3				
	5	**6**	**7**	**8**	**9**

This was the grid after the tile at (7,12) marked X had been removed. The blue tiles are the ones found by the CountWalls method and are in the NESWArray.

nextDeadEnd = CountWalls(7,12)
CountWalls {
wallCount = 0
openSide = (0,0)
NESWArray[0] = (7,13)
NESWArray[1] = (8,12)
NESWArray[2] = (7,11)
NESWArray[3] = (6,12)
}

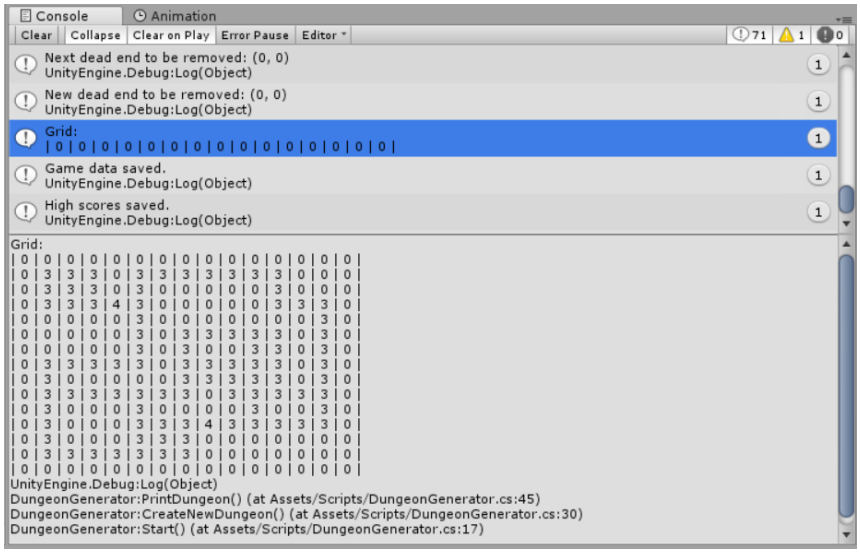
I decided to change the order of operations in the RecursiveRemove method to fix this and make sure my program checked a tile was a dead end before removing it (**figure 127, page 81**).

[Figure 127]

```
268     //The method that is called from the main CreateNewDungeon method and starts off the recursive
269     //removal of dead ends.
270     void RemoveDeadEnds () {
271         Vector2Int deadEnd = FindDeadEnds();
272         tileArray[deadEnd.x, deadEnd.y] = 0;
273         deadEnd = CountWalls(deadEnd.x, deadEnd.y);
274         RecursiveRemove(deadEnd);
275     }
276
277     //Recursivley call this method until there are no more deadEnds.
278     void RecursiveRemove (Vector2Int currentTile) {
279         Vector2Int nextDeadEnd = CountWalls(currentTile.x, currentTile.y);
280         Debug.Log("Next dead end to be removed: " + nextDeadEnd.ToString());
281         if (nextDeadEnd != Vector2Int.zero) {
282             //Close the dead end.
283             Debug.Log("Dead end removed: " + currentTile.ToString());
284             tileArray[currentTile.x, currentTile.y] = 0;
285             PrintDungeon();
286             RecursiveRemove(nextDeadEnd);
287         } else {
288             Vector2Int newDeadEnd = FindDeadEnds();
289             Debug.Log("New dead end to be removed: " + newDeadEnd.ToString());
290             if (newDeadEnd != Vector2Int.zero) {
291                 RecursiveRemove(newDeadEnd);
292             }
293         }
294     }
```

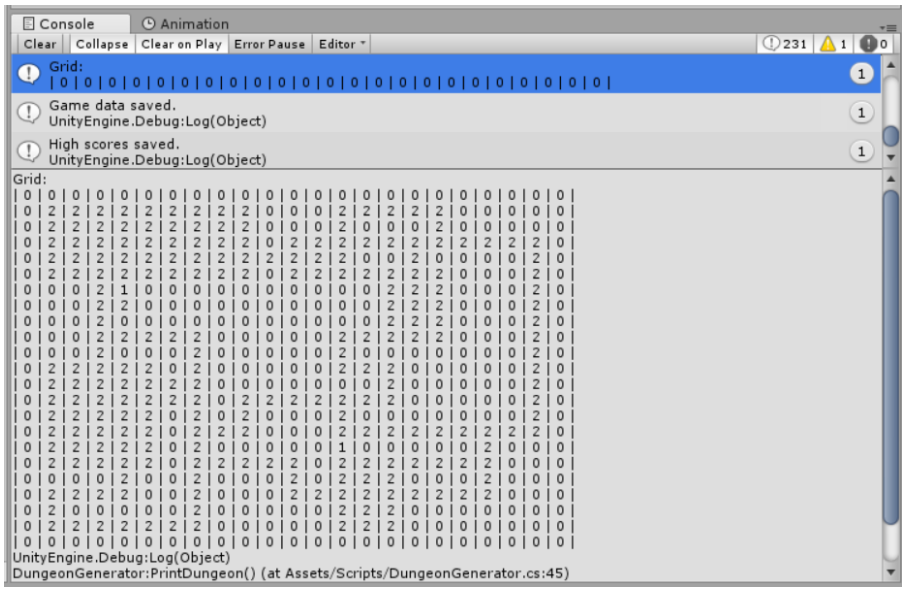
I tested again and this time the dungeon map was generated properly with all rooms connected and no dead-ends (**figure 128, page 82**).

[Figure 128]



I then tried generating a dungeon that would be created on level 5 to check bigger dungeons also worked. This test was also successful (**figure 129, page 82**).

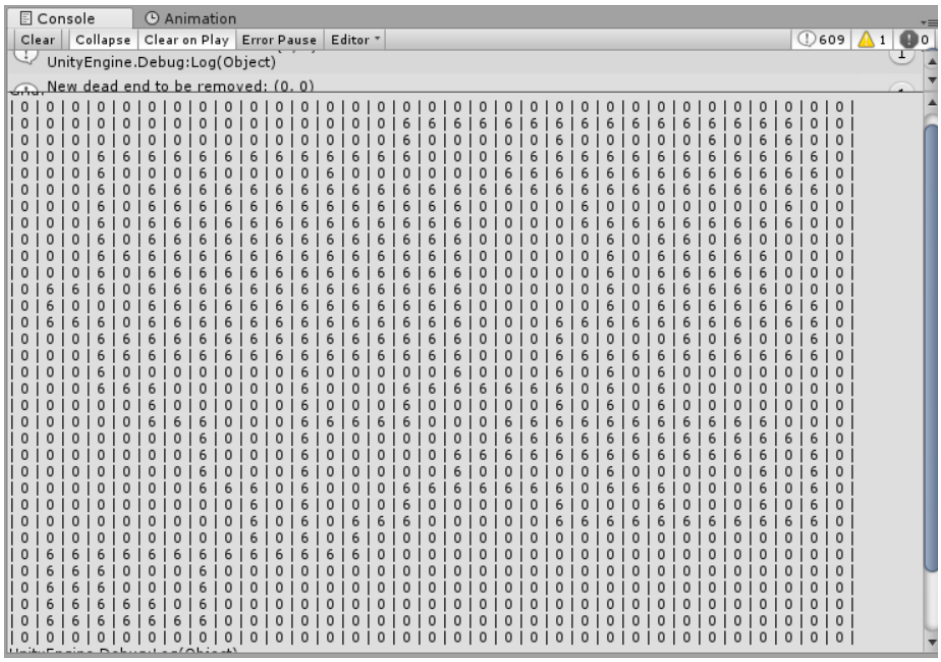
[Figure 129]



I also did a test with a level 10 dungeon (**figure 130, page 83**) to check my dungeon generator worked at even higher levels. Because the dungeon was much bigger, it took longer to generate. I did another 5 tests but this time I recorded the time to generate each dungeon to see how long it took compared to lower level dungeons (between levels 1 and 5) which generate almost instantaneously. It took between 7 and 26 seconds with an average time of 16 seconds. This is quite long so I will need to add an intermediary screen with a message that will display 'loading dungeon' to my game to let the player know the dungeon is just loading and the game hasn't frozen. I know my dungeon takes a longer time to load, the larger it is because a lot of the methods loop through the whole grid many times which takes longer, the larger the dungeon. I could teak my program to decrease the rate at which the grid increases with size as the level of the dungeon increases but I would like to test my game with my stakeholders first when I have a working prototype to find out

what the average maximum level players reach before dying, so I know how much to tweak my program.

[Figure 130]



I then removed all the debugging code from the DungeonGenerator class except for the variable for choosing the dungeon level and the method that prints a grid of the dungeon because I will need it later to check that the separate class that displays the dungeon in the scene using tile game objects, displays identical grids to the actual dungeon generated.

Displaying the dungeon using the tile grid

So far, my game can just generate a dungeon but not display it to the user, so the next step was to display the dungeon in the scene. I created a new class that I called DisplayDungeon and I attached it to the Tile Map game object in the scene which I renamed DungeonTiles. First, the DisplayDungeon class will need access to the dungeon grid from the DungeonGenerator class, but it will not access the grid directly. After the DungeonGenerator class has finished generating a dungeon, it should hand the grid to the PersistentGameData class, so it can be saved and reloaded. All other classes should then be able to access the grid through the PersistentGameData class. I edited the PersistentGameData class (**figure 131, page 84**) and the GameData class (**figure 132, page 85**) so that it would be able to store the grid.

[Figure 131]

```
using System.Collections;
using System.Collections.Generic;
using System;
using UnityEngine;

public class PersistentGameData : MonoBehaviour {

    public int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
    public string playerName;
    public Vector3 playerPosSave;
    public int[] operatorCodesArray;
    public int[,] dungeonMapSave;

    //When this object is instantiated, this class will load the saved data from the file or create a set of new data
    //if there is no file and this is a new game.
    void Awake () {
        bool dataLoaded = LoadGameData();
        if (!dataLoaded) {
            operatorCodesArray = new int[4];
            playerPosSave = new Vector3(0,0,0);
            dungeonLevel = 1;
            playerLivesSave = 3;
            potions = 3;
        }
    }

    //Before this object is destroyed, save the game data to the file.
    void OnDisable () {
        SaveGameData();
    }
    //This method is run before a new scene is loaded to save the data of the previous scene.
    void SaveGameData () {
        GameData SavedGameData = new GameData();
        SavedGameData.dungeonLevel = dungeonLevel;
        SavedGameData.rangeMax = rangeMax;
        SavedGameData.rangeMin = rangeMin;
        SavedGameData.maxMoves = maxMoves;
        SavedGameData.playerLivesSave = playerLivesSave;
        SavedGameData.playerName = playerName;
        SavedGameData.potions = potions;
        SavedGameData.operatorCodesArray = operatorCodesArray;
        SavedGameData.playerPosSaveX = playerPosSave.x;
        SavedGameData.playerPosSaveY = playerPosSave.y;
        SavedGameData.playerPosSaveZ = playerPosSave.z;
        SavedGameData.dungeonMapSave = dungeonMapSave;
        SaveGameSystem.SaveGame(SavedGameData, "SavedGameData");
        Debug.Log("Game data saved.");
    }
    //This method uses the SaveGameSystem to load the data from the file and returns true if it finds the file.
    bool LoadGameData () {
        if (SaveGameSystem.DoesSaveGameExist ("SavedGameData")) {
            GameData SavedGameData = SaveGameSystem.LoadGame("SavedGameData") as GameData;
            dungeonLevel = SavedGameData.dungeonLevel;
            rangeMax = SavedGameData.rangeMax;
            rangeMin = SavedGameData.rangeMin;
            maxMoves = SavedGameData.maxMoves;
            playerLivesSave = SavedGameData.playerLivesSave;
            playerName = SavedGameData.playerName;
            potions = SavedGameData.potions;
            operatorCodesArray = SavedGameData.operatorCodesArray;
            dungeonMapSave = SavedGameData.dungeonMapSave;

            //The vector of the player's position is created out of the induvidual coordinates for each axis.
            playerPosSave = new Vector3(SavedGameData.playerPosSaveX, SavedGameData.playerPosSaveY, SavedGameData.playerPosSaveZ);
            Debug.Log("Game data loaded.");
            return true;
        }
        return false;
    }
}
```

[Figure 132]

```
74 //This is a separate private class that inherits from SaveGame and is used by PersistentGameData to store the game data
75 //in a serialized permentant file.
76 [Serializable]
77 class GameData : SaveGame {
78
79     public int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
80     public string playerName;
81     public float playerPosSaveX, playerPosSaveY, playerPosSaveZ;
82     public int[] operatorCodesArray;
83     public int[,] dungeonMapSave;
84
85     //The vector of the player's position is saved separately as three floats of each separate coordinate because Vector3
86     //is not a standard data type.
87 }
```

I then edited the DungeonGenerator class so that it would pass the tileArray variable to the PersistentGameData class after the dungeon had been generated (**figure 133, page 85**).

[Figure 133]

```
20     //This method is called whenever a new dungeon needs to be created.
21     void CreateNewDungeon () {
22         newRegionNumber = 1;
23         CreateGrid(testLevelValue); //pgd.dungeonLevel will be used instead of testLevelValue when testing is done.
24         PlaceRooms(testLevelValue); //pgd.dungeonLevel will be used instead of testLevelValue when testing is done.
25         frontierTilesList = new List<Vector2Int>();
26         GenerateCorridors();
27         connectors = new List<int[]>();
28         ConnectAllRegions();
29         RemoveDeadEnds();
30         PrintDungeon();
31         pgd.dungeonMapSave = tileArray;
32     }
```

I then started writing the DisplayDungeon class, first I gave it access to the wall and floor prefabs, so it could instantiate these different types of tiles. I would attach the prefab of these tiles to the class in Unity's editor. I also gave it access to the PersistentGameData class, so it could retrieve the dungeon grid and use it to place tiles at the correct coordinates. I created a new method called DisplayTileGrid that would loop through the dungeon map from the PersistentGameData class and place tiles according to the region number of each coordinate (**figure 134, page 85**).

[Figure 134]

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.Tilemaps;
5
6 public class DisplayDungeon : MonoBehaviour {
7
8     public Tile wall, floor;
9
10    private PersistentGameData pgd;
11    private Tilemap tileMap;
12
13    //Finds the PersistentGameData class and the Tilemap component of the game object this class is attached to.
14    void Start () {
15        pgd = GameObject.FindObjectOfType<PersistentGameData>();
16        tileMap = GetComponent<Tilemap>();
17    }
```

```
19      //Loops through all tiles and sets the correct tile at the correct position on the tile map.
20      public void DisplayTileGrid () {
21          float totalElements = (float)pgd.dungeonMapSave.Length;
22          int gridLength = Mathf.RoundToInt(Mathf.Sqrt(totalElements)) - 1;
23          int[,] dungeonGrid = pgd.dungeonMapSave;
24          Vector3Int currentCoords = new Vector3Int(0,0,0);
25          for (int y = 0; y <= gridLength; y++) {
26              for (int x = 0; x <= gridLength; x++) {
27                  currentCoords.x = x;
28                  currentCoords.y = y;
29                  if (dungeonGrid[x,y] == 0) {
30                      tileMap.SetTile(currentCoords, wall);
31                  } else {
32                      tileMap.SetTile(currentCoords, floor);
33                  }
34              }
35          }
36      }
37
38  }
```

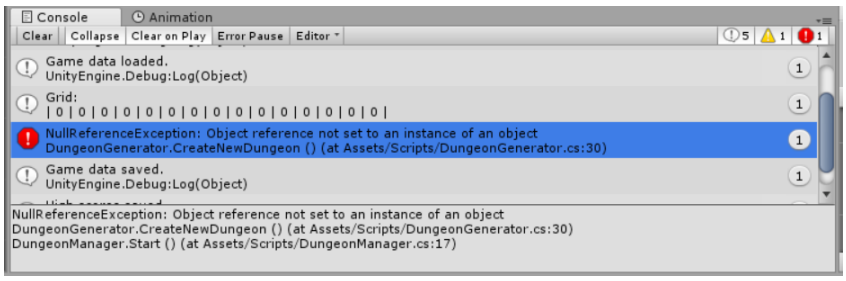
The DisplayTileGrid method must only be called after the dungeon has been generated or loaded from a previous save. To coordinate the different classes, I decided to create a new game object called DungeonManager (**figure 135, page 86**) that would manage what was displayed on the dungeon scene. I created a new class called DungeonManager which would coordinate the other classes as well as the dialog box in this scene and manage what was displayed with the user interface. At the moment I just made a method for the DungeonManager that will coordinate the generating and loading of the dungeon. I will add more functionality as I progress development.

[Figure 135]

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class DungeonManager : MonoBehaviour {
6
7      private PersistentGameData pgd;
8      private DungeonGenerator dungeonGen;
9      private DisplayDungeon displayDungeon;
10
11     //Finds the other classes on the level and manages when a new dungeon is generated.
12     void Start () {
13         pgd = GameObject.FindObjectOfType<PersistentGameData> ();
14         dungeonGen = GameObject.FindObjectOfType<DungeonGenerator> ();
15         displayDungeon = GameObject.FindObjectOfType<DisplayDungeon> ();
16         if (pgd.dungeonMapSave == null) {
17             dungeonGen.CreateNewDungeon ();
18         }
19         displayDungeon.DisplayTileGrid ();
20     }
21 }
```

I then tested this scene to check that the dungeon that was generated was displayed properly. The test was unsuccessful as nothing was displayed and I got this error (**figure 136, page 86**).

[Figure 136]



It seemed like the dungeonMapSave variable of the PersistentGameData class could not being accessed for some reason. I tried assigning the dungeonMapSave variable to other values. This didn't work so I tried initialising the variable in its own class which was causing some of the problem. In case this happened to any other variables, I initialised all the other variables of this class in the Awake method (**figure 137, page 87**). I gave it default values which meant I also had to change the condition for generating a new dungeon in the DungeonManager class (**figure 138, page 87**).

[Figure 137]

```
14    //When this object is instantiated, this class will load the saved data from the file or create a set of new data
15    //if there is no file and this is a new game.
16    void Awake () {
17        bool dataLoaded = LoadGameData();
18        if (!dataLoaded) {
19            operatorCodesArray = new int[4];
20            dungeonMapSave = new int[1,1] {{0}};
21            playerPosSave = new Vector3(0,0,0);
22            dungeonLevel = 1;
23            playerLivesSave = 3;
24            potions = 3;
25            rangeMin = 0;
26            rangeMax = 0;
27            maxMoves = 0;
28            playerName = "";
29        }
30    }
```

[Figure 138]

```
11    //Finds the other classes on the level and manages when a new dungeon is generated.
12    void Start () {
13        pgd = GameObject.FindObjectOfType<PersistentGameData> ();
14        dungeonGen = GameObject.FindObjectOfType<DungeonGenerator> ();
15        displayDungeon = GameObject.FindObjectOfType<DisplayDungeon> ();
16        if (pgd.dungeonMapSave.Length == 1) {
17            dungeonGen.CreateNewDungeon ();
18        }
19        displayDungeon.DisplayTileGrid ();
20    }
21 }
```

I then tested again, and I still got the same error message, so I decided to make all public variables of the PersistentGameData class static (**figure 139, page 87**) since there should only be one instance of all the variables of the class so that all the other classes can access and modify the same data. I also did the same to the PersistentHighScores class (**figure 140, page 87**) because it should work in the same way. I had to change the references to the persistent data classes in all other scripts so that they now accessed the static variables.

[Figure 139]

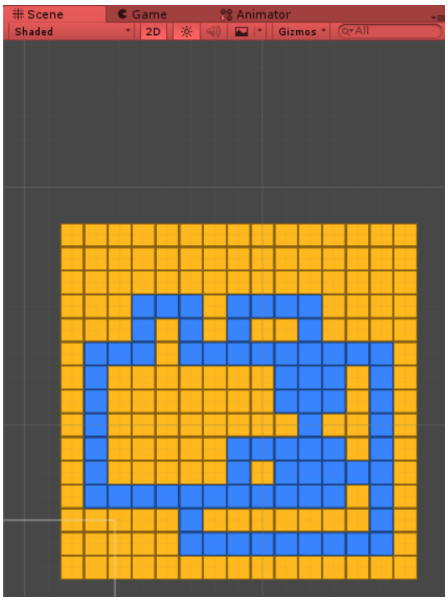
```
6 public class PersistentGameData : MonoBehaviour {
7
8     public static int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
9     public static string playerName;
10    public static Vector3 playerPosSave;
11    public static int[] operatorCodesArray;
12    public static int[,] dungeonMapSave;
```

[Figure 140]

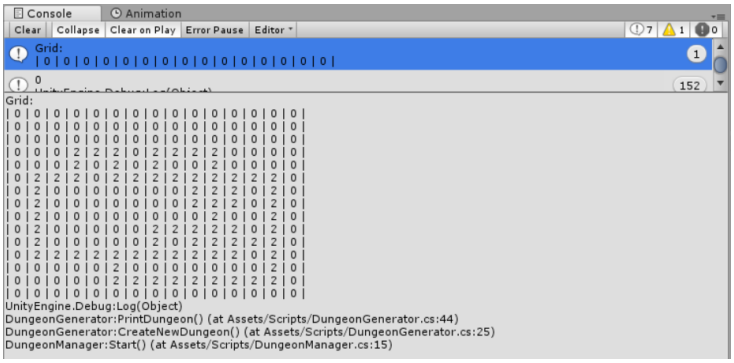
```
6 public class PersistentHighScores : MonoBehaviour {
7
8     public static int maxFloorsCleared, enemiesDefeated, bossesDefeated;
```

When I tested again, the dungeon was finally displayed using the tile game objects (**figure 141, page 88**). I checked that it was identical to what the Dungeon Generator printed in the console (**figure 142, page 88**), and it was.

[Figure 141]

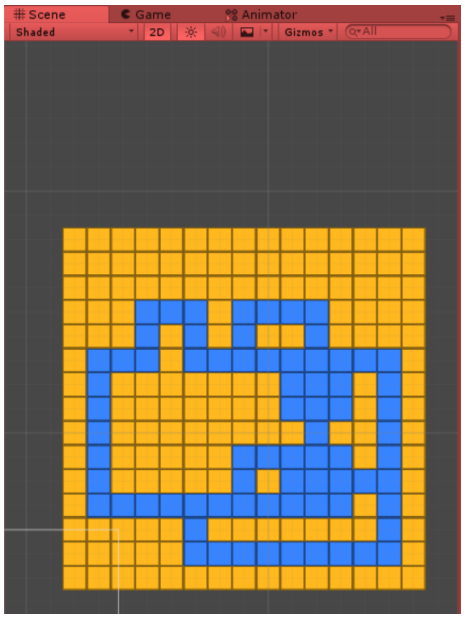


[Figure 142]



While debugging I had commented out the condition in the DungeonManager class that would control whether a new dungeon was generated so I could check that whenever a new dungeon was generated, it was displayed correctly. I added back in the condition to test whether the dungeon that was just generated could be displayed after the grid had been saved in the PersistentGameData class. The test was successful as no new dungeons were generated and the previous dungeon was displayed (**figure 143, page 88**).

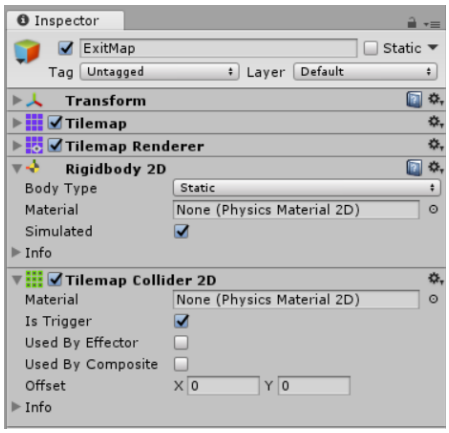
[Figure 143]



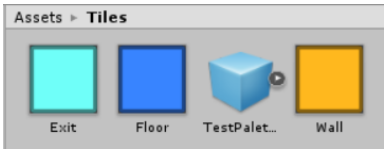
The only parts missing from the dungeon were now the entrance and the exit. The exit should be different coloured tile which can be identified as the exit and the entrance will be a position in the dungeon where the character starts. I first created a new light blue tile to be the exit and it will trigger the next dungeon level to load (**figure 145, page 89**). I created a new Tilemap game object called ExitMap (**figure 144, page 89**) to hold this tile which will be rendered above the DungeonTiles TileMap. I did this because the Exit tile will use a trigger collider to detect if the player walks on this

tile which is a different type of collider from the wall tiles which don't allow the player to walk on them.

[Figure 144]



[Figure 145]



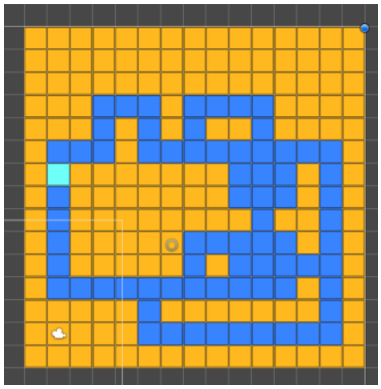
To place the exit tile in the dungeon, the DisplayDungeon class will pick a random floor tile to become an exit tile and another to become the starting position. I edited the DisplayDungeon class to place the exit tile and find a starting position (**figure 146, page 89**).

[Figure 146]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Tilemaps;

public class DisplayDungeon : MonoBehaviour {

    public Tile wall, floor, exit;
    public GameObject ExitMapObject;
    public Vector3Int startingPosition;

    private Tilemap tileMap, exitMap;

    //Finds the Tilemap components of the game object this class is attached to and the other tile map game objects.
    void Start () {
        tileMap = GetComponent<Tilemap>();
        exitMap = ExitMapObject.GetComponent<Tilemap>();
    }

    //Loops through all tiles and sets the correct tile at the correct position on the tile map.
    public void DisplayTileGrid () {
        float totalElements = (float)PersistentGameData.dungeonMapSave.Length;
        int gridLength = Mathf.RoundToInt(Mathf.Sqrt(totalElements)) - 1;
        int[,] dungeonGrid = PersistentGameData.dungeonMapSave;
        int floorArea = gridLength * gridLength;
        Vector3Int[] floorTileCoords = new Vector3Int[floorArea];
        int newTile = 0;
        Vector3Int currentCoords = new Vector3Int(0,0,0);
        for (int y = 0; y <= gridLength; y++) {
            for (int x = 0; x <= gridLength; x++) {
                currentCoords.x = x;
                currentCoords.y = y;
                if (dungeonGrid[x,y] == 0) {
                    tileMap.SetTile(currentCoords, wall);
                } else {
                    tileMap.SetTile(currentCoords, floor);
                    floorTileCoords[newTile] = currentCoords;
                    newTile++;
                }
            }
        }
        //Picks the position of the dungeon exit and the player's starting position.
        int randomTile = Random.Range(0, floorTileCoords.Length);
        exitMap.SetTile(floorTileCoords[randomTile], exit);
        int entrancePosition;
        do {
            entrancePosition = Random.Range(0, floorTileCoords.Length);
        } while (entrancePosition == randomTile);
        startingPosition = floorTileCoords[entrancePosition];
    }
```

I then tested the game to check that that the exit tile was being placed correctly on a floor tile and the starting position was also on a floor tile. The exit was placed correctly (**figure 147, page 90**) but the startingPosition variable had the wrong value because it showed the coordinates (0,0,0) (**figure 148, page 90**) which is a wall in my dungeon.

[Figure 147]



[Figure 148]

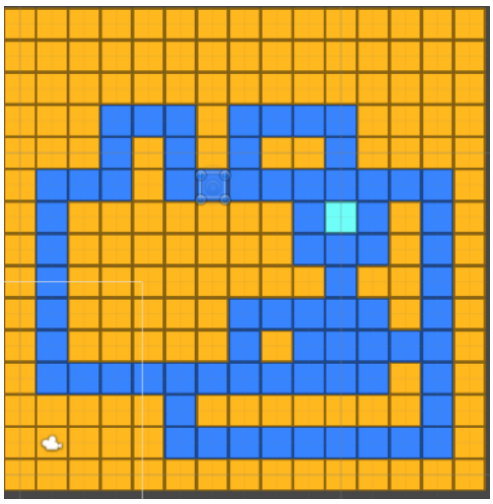
I tried the test again and this time the exit tile was also given the coordinates (0,0,0) as well as the starting position so I decided to print what was in the floorTileCoords array to find out where this coordinate was coming from. I found out that when I was finding a random coordinate in the array, I was picking a random coordinate from the whole array which wasn't completely full of wall tile coordinates. At least half of the array was the coordinate (0,0,0) so I changed the code of the DisplayTileGrid method to only find tiles in the array that were floor tiles (**figure 149, page 90**).

[Figure 149]

```
42        //Picks the position of the dungeon exit and the player's starting position.
43        int randomTile = Random.Range(0, newTile);
44        exitMap.SetTile(floorTileCoords[randomTile], exit);
45        int entrancePosition;
46        do {
47            entrancePosition = Random.Range(0, newTile);
48        } while (entrancePosition == randomTile);
49        startingPosition = floorTileCoords[entrancePosition];
50    }
```

I tested again a few times and this time it worked (**figure 150, page 90**), and no coordinates were (0,0,0).

[Figure 150]



I then needed to create a way of saving the position of the exit, otherwise it would change position every time the game was loaded. I added a Vector3 variable to the PersistentGameData class for storing the position of the exit tile called exitPosSave. I edited the rest of the class to make sure it was saved along with the variables. Since Vector3 is not a standard data type, I had to save it the same way as the variable PlayerPosSave by saving the individual coordinates of each axis. I added a reference to the new variable in the DisplayDungeon class to save the coordinates it generates there (**figure 151, page 91**).

[Figure 151]

```
42          //Picks the position of the dungeon exit and the player's starting position.
43          int randomTile = Random.Range(0, newTile);
44          exitMap.SetTile(floorTileCoords[randomTile], exit);
45          PersistentGameData.exitPosSave = (Vector3)floorTileCoords[randomTile];
46          int entrancePosition;
47          do {
48              entrancePosition = Random.Range(0, newTile);
49          } while (entrancePosition == randomTile);
50          startingPosition = floorTileCoords[entrancePosition];
51      }
```

Now I just needed to change my code so that it only creates new coordinates for the exit after a new dungeon has been generated and can place a dungeon tile from the coordinates loaded from memory. I edited the code again and I added a Boolean parameter to the DisplayTileGrid method that would tell it whether a new dungeon had just been generated (**figure 152, page 91**).

[Figure 152]

```
42          //Picks the position of the dungeon exit and the player's starting position if a new dungeon is generated,
43          //otherwise the exit is placed at the coordinates loaded from memory.
44          if (isNewDungeon == true) {
45              int exitPosition;
46              exitPosition = Random.Range(0, newTile);
47              exitMap.SetTile(floorTileCoords[exitPosition], exit);
48              PersistentGameData.exitPosSave = (Vector3)floorTileCoords[exitPosition];
49              int entrancePosition;
50              do {
51                  entrancePosition = Random.Range(0, newTile);
52                  startingPosition = floorTileCoords[entrancePosition];
53              } while (exitPosition == entrancePosition);
54          } else {
55              Vector3Int exitCoords;
56              exitCoords = Vector3Int.RoundToInt(PersistentGameData.exitPosSave);
57              exitMap.SetTile(exitCoords, exit);
58          }
59      }
```

I also edited the code of the Dungeon Manager class so that it could tell the DisplayTileGrid method whether this was a new dungeon (**figure 153, page 91**).

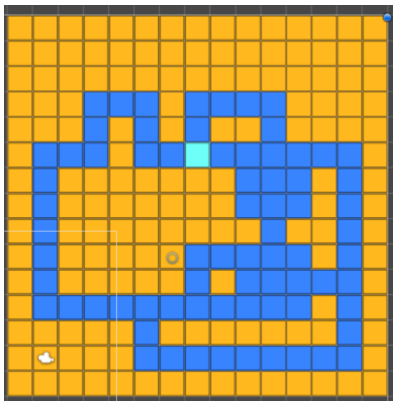
[Figure 153]

```
10      //Finds the other classes on the level and manages when a new dungeon is generated.
11      void Start () {
12          dungeonGen = GameObject.FindObjectOfType<DungeonGenerator> ();
13          displayDungeon = GameObject.FindObjectOfType<DisplayDungeon> ();
14          if (PersistentGameData.dungeonMapSave.Length == 1) {
15              dungeonGen.CreateNewDungeon ();
16              displayDungeon.DisplayTileGrid (true);
17          } else {
18              displayDungeon.DisplayTileGrid (false);
19          }
20      }
21  }
```

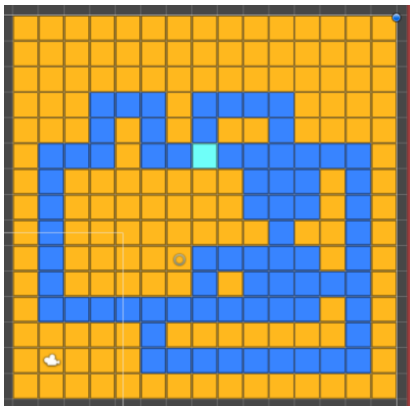
I then tested this to make sure the exit tile was placed in the same place each time the game was loaded (**figure 154, page 92**). I didn't test to see if the startingPosition was saved because it doesn't need to be as it just used to initially place the player in the dungeon. The player's position will later

be saved by its own game object. The first test showed the exit in a suitable position. I tested again to check whether the exit would be in the same place when I loaded the game again. This test was also successful (**figure 155, page 92**).

[Figure 154]



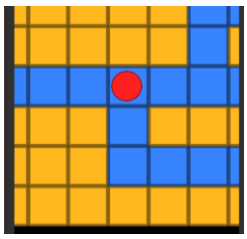
[Figure 155]



Creating the player character

Now the only things still missing from the scene were the player and the user interface. To create the character that the player can control to traverse the dungeon, I started by creating a test version of the player character which is represented as a red circle (**figure 156, page 92**). I gave the player character 2D rigid body and capsule collider components so that it could interact with the colliders on the wall tiles of the dungeon map. I also made the player kinematic as it is controlled by the user.

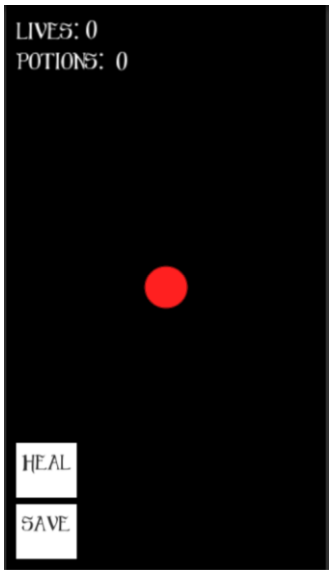
[Figure 156]



Creating the dungeon scene interface

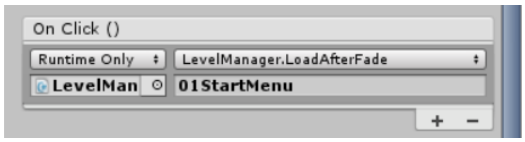
I then created a basic version of the rest of the UI. This included a display of the player's potions and the number of lives they have left as well as a save button to save their progress which takes the player to the StartMenu scene. The heal button will use one potion to give the player back all their lives which is currently 3 but it could change depending on when I test this with my stakeholders and they find the number of lives too easy or difficult. The red circle is the player character which will always be centre screen (**figure 157, page 93**). I also added a fader game object to this scene, but I left it disabled until this scene is finished.

[Figure 157]



I first added functionality to the save button by making it call the LevelManager class's LoadAfterFade and making it load the start menu scene which would trigger the PersistentGameData class to save the player's progress (**figure 158, page 93**).

[Figure 158]



I added more code to the DungeonManager class to handle displaying the correct numbers of lives and potions as well as the function of healing the player (**figure 159, page 93**). I made sure that the saved values of the lives and the potions were displayed when the scene is first loaded. I then created a new method for when the player presses the heal button (**figure 160, page 94**).

[Figure 159]

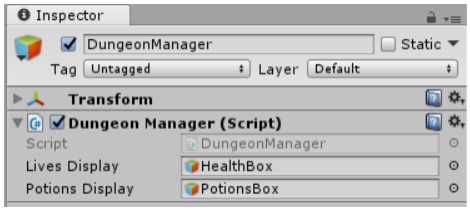
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class DungeonManager : MonoBehaviour {

    public GameObject livesDisplay, potionsDisplay;

    private DungeonGenerator dungeonGen;
    private DisplayDungeon displayDungeon;
    private Text lives, potions;

    //Finds the other classes on the Level and manages when a new dungeon is generated.
    void Start () {
        dungeonGen = GameObject.FindObjectOfType<DungeonGenerator> ();
        displayDungeon = GameObject.FindObjectOfType<DisplayDungeon> ();
        if (PersistentGameData.dungeonMapSave.Length == 1) {
            dungeonGen.CreateNewDungeon ();
            displayDungeon.DisplayTileGrid (true);
        } else {
            displayDungeon.DisplayTileGrid (false);
        }
        //Displays the number of lives and potions the palyer has.
        lives = livesDisplay.GetComponent<Text>();
        potions = potionsDisplay.GetComponent<Text>();
        lives.text = PersistentGameData.playerLivesSave.ToString();
        potions.text = PersistentGameData.potions.ToString();
    }
```

[Figure 160]

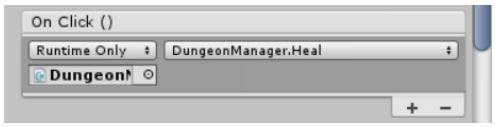
```
31    //When the player presses the heal button, one potion is used to give the player full health.
32    public void Heal () {
33        PersistentGameData.potions -= 1;
34        PersistentGameData.playerLivesSave = 3;
35        lives.text = PersistentGameData.playerLivesSave.ToString();
36        potions.text = PersistentGameData.potions.ToString();
37    }
38 }
```

I attached the HealthBox and PotionsBox to the DungeonManager class (**figure 161, page 94**) and I attached the new Heal method to the heal button in Unity (**figure 162, page 94**).

[Figure 161]

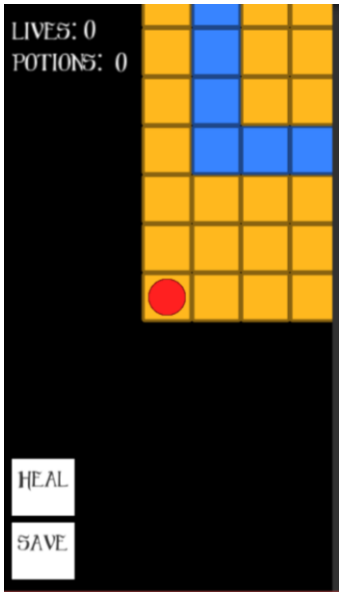


[Figure 162]



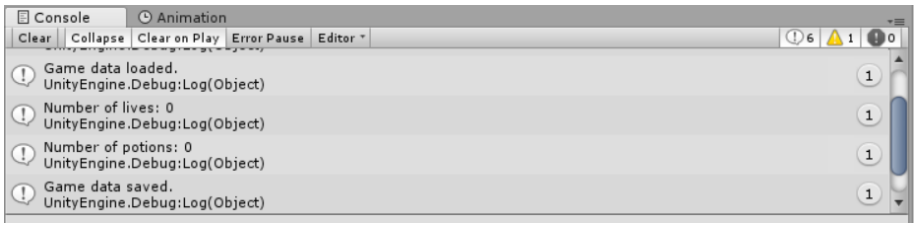
I then tested my game to make sure the numbers of lives and potions were displayed. The test was unsuccessful as both were displayed as zero (**figure 163, page 94**).

[Figure 163]



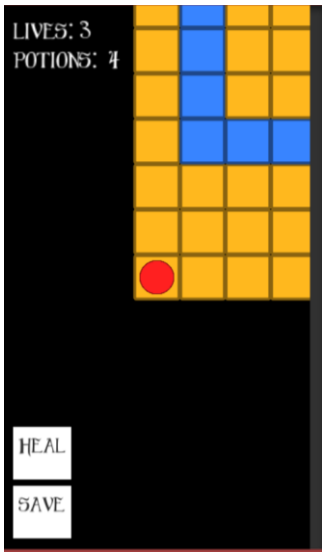
I tried printing the values in the console to see what the saved values of both of the variables were. I found out that both of the saved values were 0 so they were being displayed properly but had no values (**figure 164, page 95**).

[Figure 164]



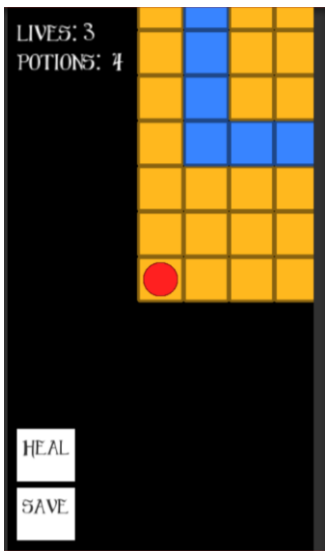
I added some temporary code to assign values to these variables that I would remove after running the game to test that they were saved. I then ran the game with the temporary code to assign values the values (**figure 165, page 95**).

[Figure 165]



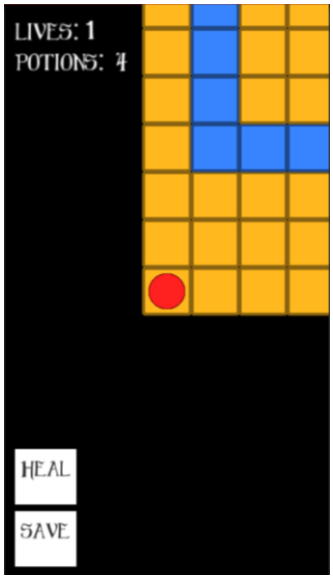
When I ran the game again after I had removed the temporary code, the values had been saved and were now being displayed so the test was successful (**figure 166, page 95**).

[Figure 166]

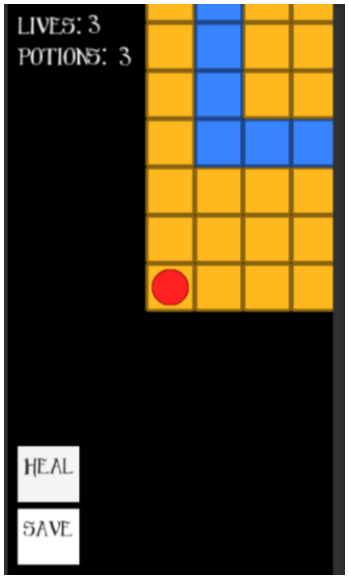


I then tested the heal button to make sure it healed the player. Before I tested I reduced the player's lives to 1 so I could check that the number of lives was being changed (**figure 167, page 96**). The test was successful as the number of potions was decreased by 1 and the lives were now 3 (**figure 168, page 96**).

[Figure 167]

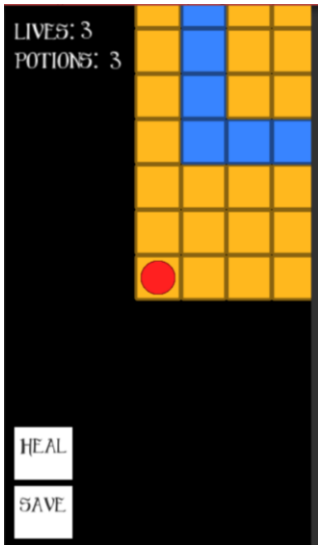


[Figure 168]



I tested again to check the values of the lives and the potions were saved after healing when I reloaded the game. This test was successful as both values had been saved and were loaded properly (**figure 169, page 96**).

[Figure 169]

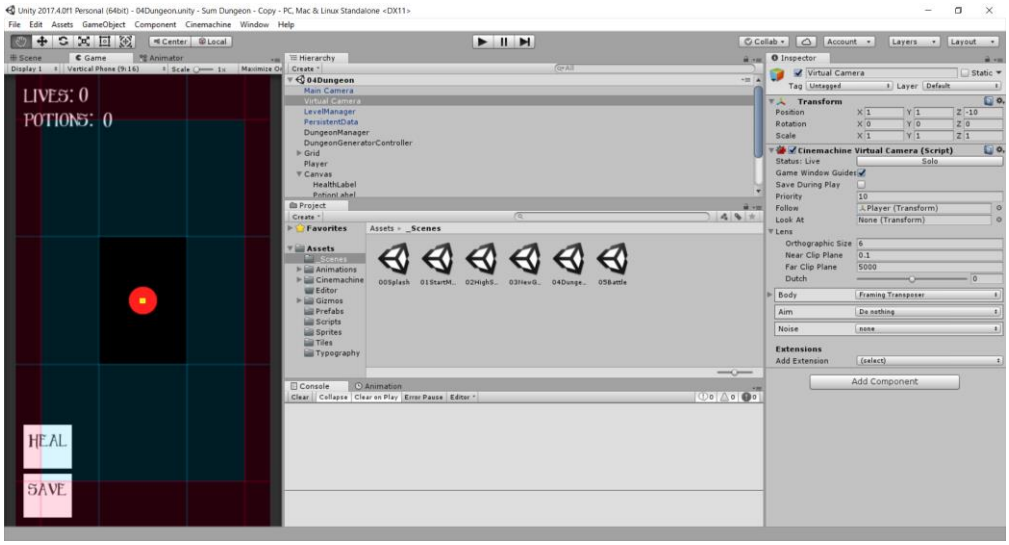


I then removed the debugging code from the DungeonManager script.

Importing Cinemachine and configuring the scene camera

I added the Cinemachine asset to my project from the unity asset store to control the camera in the Dungeon scene. I created a new 2D Cinemachine virtual camera game object that would now control the movement of the camera and keep it focused on the player character (**figure 170, page 97**). So, when the player moves, the camera will also move.

[Figure 170]



Developing the player character's movement

Then I started making the class that controls the movement of the character. I created a new class called player movement. First, I created a new class that only the PlayerMovement class (**figure 171, page 97**) could access called Node (**figure 172, page 99**). This is a structure that would hold information about the nodes such as their coordinates on the dungeon grid, their parent Node and their f and g costs. I then programmed the A* algorithm to control the pathfinding of the player character in the dungeon. I used the nodes with the method AStarPathfinding to find a path from the player's position to the position they selected in the dungeon. I also created other methods to break this method into smaller parts such as FindNeighbours and CalculateFCost methods. I also created a method called FindNodeInList to find any duplicate nodes in the lists. After creating the methods to look through all the nodes, I created the CreatePath method to find the coordinates of each node on the path and create a list of them starting from the player character's position to the goal position. The last part was displaying the player walk to the goal, so I created an AnimateWalk method to move the player character, one tile at a time to the goal.

[Figure 171]

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class PlayerMovement : MonoBehaviour {
6
7     public GameObject mainCamera;
8     public float movementSpeed = 0.5f;
9     public Vector2Int goal;
10    public List<Vector3Int> path;
11
12    private List<Node> openNodes, visitedNodes;
13    private int[,] dungeonGrid;
14    private int gridLength;
15    private Camera playerCam;
16    private bool characterMoving;
17    private float nextStepTime;
18    private int count;
19
20    //Finds the camera, dungeon grid and gridLength.
21    void Start () {
22        playerCam = mainCamera.GetComponent<Camera>();
23        dungeonGrid = PersistentGameData.dungeonMapSave;
24        float totalElements = (float)PersistentGameData.dungeonMapSave.Length;
25        gridLength = Mathf.RoundToInt(Mathf.Sqrt(totalElements)) - 1;
26        characterMoving = false;
27        nextStepTime = 0f;
28        count = 0;
29    }
```

```
    //Checks every frame for the player's touch on screen, if the player selects a floor tile in the dungeon
    //the player character will travel there.
    void Update () {
        if (characterMoving == true) {
            AnimateWalk();
        } else if (Input.touchCount == 1) {
            Touch fingerPos = Input.GetTouch(0);
            Vector2 tileTouchedPos = playerCam.ScreenToWorldPoint(fingerPos.position);
            goal = Vector2Int.RoundToInt(tileTouchedPos);
            if (goal.x > 0 && goal.x < gridLength && goal.y > 0 && goal.y < gridLength) {
                if (dungeonGrid[goal.x, goal.y] != 0) {
                    openNodes.Clear();
                    visitedNodes.Clear();
                    path.Clear();
                    AStarPathfinding();
                    characterMoving = true;
                }
            }
        }
    }
    //Animate the player character moving through the dungeon.
    void AnimateWalk () {
        if (Time.time > nextStepTime && count != path.Count) {
            nextStepTime += movementSpeed;
            gameObject.transform.position = path[count];
            count++;
        } else {
            characterMoving = false;
            count = 0;
            nextStepTime = 0f;
        }
    }
    //Pathfinds to the goal coordinates using the A* algorithm.
    void AStarPathfinding () {
        Vector2Int startCoords = Vector2Int.RoundToInt(gameObject.transform.position);
        Node startNode = new Node(startCoords, null, 0, 0);
        openNodes = new List<Node>();
        openNodes.Add(startNode);
        visitedNodes = new List<Node>();
        Node currentNode = startNode;
        //While the path has not reached the goal.
        while (currentNode.coords != goal) {
            Vector2Int[] neighbours = FindNeighbours(currentNode.coords);
            foreach (Vector2Int n in neighbours) {
                if (n != Vector2Int.zero) {
                    Node newNode = new Node(n, currentNode, 0, currentNode.gCost + 1);
                    int FCost = CalculateFCost(newNode);
                    newNode.fCost = FCost;
                    Node duplicate = FindNodeInList(newNode, false);
                    if (duplicate != null) {
                        duplicate = FindNodeInList(newNode, true);
                        if (duplicate != null && newNode.fCost < duplicate.fCost) {
                            openNodes.Add(newNode);
                        }
                    }
                }
            }
            visitedNodes.Add(currentNode);
            openNodes.Remove(currentNode);
            foreach (Node m in openNodes) {
                //Find the node with least fCost to be next current Node.
                if (m.fCost <= currentNode.fCost) {
                    currentNode = m;
                }
            }
        }
        visitedNodes.Add(currentNode);
        CreatePath();
    }

    //Creates a list of the coordinates of each tile on the path found by the A* algorithm.
    void CreatePath () {
        Node currentNode = visitedNodes[visitedNodes.Count-1];
        while (currentNode.parentNode != null) {
            Vector3Int nodeCoords = new Vector3Int(currentNode.coords.x, currentNode.coords.y, 0);
            path.Add(nodeCoords);
            currentNode = currentNode.parentNode;
        }
        path.Reverse();
    }
```

```
114     //Finds a node with specific coordinates in a specific list and return it, otherwise return null.
115     Node FindNodeInList (Node node, bool openList) {
116         //Finds any node that has the same coordinates as the node given.
117         System.Predicate<Node> duplicateNodeFinder = (Node n) => {return n.coords == node.coords; };
118         //If I want to search the openNodes list.
119         if (openList == true) {
120             Node foundNode1 = openNodes.Find(duplicateNodeFinder);
121             return foundNode1;
122         //If I want to search the visitedNodes list.
123         } else {
124             Node foundNode2 = visitedNodes.Find(duplicateNodeFinder);
125             return foundNode2;
126         }
127     }
128 
129     //Calculates the fCost of a node using the Manhattan distance heuristic.
130     int CalculateFCost (Node node) {
131         int xCoord = node.coords.x;
132         int yCoord = node.coords.y;
133         int hCost = Mathf.Abs(xCoord-goal.x) + Mathf.Abs(yCoord-goal.y);
134         int fCost = node.gCost + hCost;
135         return fCost;
136     }
138     //Find the given tile's neighbours.
139     Vector2Int[] FindNeighbours (Vector2Int coords) {
140         int xCoord = coords.x;
141         int yCoord = coords.y;
142         Vector2Int[] NESWArray = new Vector2Int[4];
143         NESWArray [0] = new Vector2Int (xCoord, yCoord + 1);
144         NESWArray [1] = new Vector2Int (xCoord + 1, yCoord);
145         NESWArray [2] = new Vector2Int (xCoord, yCoord - 1);
146         NESWArray [3] = new Vector2Int (xCoord - 1, yCoord);
147         //Get rid of coordinates that are walls or outside the grid.
148         for (int i = 0; i <= 3; i++) {
149             if (NESWArray [i].x > gridLength || NESWArray [i].x < 0) {
150                 NESWArray [i] = Vector2Int.zero;
151             } else if (NESWArray [i].y > gridLength || NESWArray [i].y < 0) {
152                 NESWArray[i] = Vector2Int.zero;
153             } else if (dungeonGrid[NESWArray[i].x, NESWArray[i].y] == 0) {
154                 NESWArray[i] = Vector2Int.zero;
155             }
156         }
157         return NESWArray;
158     }
159 
160 }
```

[Figure 172]

```
162 //The class used to create nodes, which keeps a Node's data in one object.
163 class Node : MonoBehaviour {
164 
165     public Vector2Int coords;
166     public Node parentNode;
167     public int fCost, gCost;
168 
169     //A node keeps track of it's coordinates, it's parent node and it's f and g costs.
170     public Node (Vector2Int coOrds, Node parent, int costF, int costG) {
171         coords = coOrds;
172         parentNode = parent;
173         fCost = costF;
174         gCost = costG;
175     }
176 }
```

Testing the player spawned on the correct tile

I started to test the player's movement. When I ran the game, the player stayed where it was instead of starting on a floor tile of the dungeon. I realised this was because I hadn't connected the startPosition from the DisplayDungeon class. I edited the DisplayDungeon class to have access to the player game object, so it could set the player's position (**figure 173, page 100**).

[Figure 173]

```
50            do {
51                entrancePosition = Random.Range(0, newTile);
52                startingPosition = floorTileCoords[entrancePosition];
53                player.transform.position = startingPosition;
54                PersistentGameData.playerPosSave = startingPosition;
55            } while (exitPosition == entrancePosition);
```

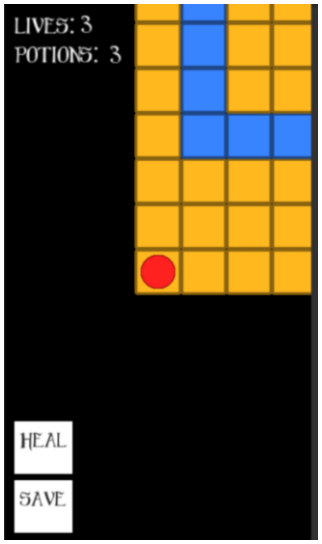
I also hadn't added the code to the PlayerMovement class to save the player's position, so I added that as well. I made it so that after the player character had reached a goal after walking, it's position would be saved to the PersistentGameData class (**figure 174, page 100**).

[Figure 174]

```
53    //Animate the player character moving through the dungeon.
54    void AnimateWalk () {
55        if (Time.time > nextStepTime && count != path.Count) {
56            nextStepTime += movementSpeed;
57            gameObject.transform.position = path[count];
58            count++;
59        } else {
60            PersistentGameData.playerPosSave = gameObject.transform.position;
61            characterMoving = false;
62            count = 0;
63            nextStepTime = 0f;
64        }
65    }
```

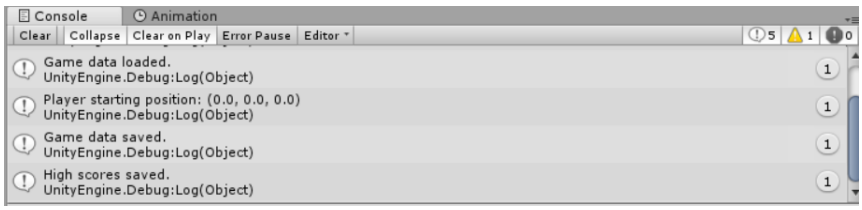
I tested my game again, but I was still unsuccessful as the player just moved to the coordinates (0,0,0) which is also a wall (**figure 175, page 100**).

[Figure 175]



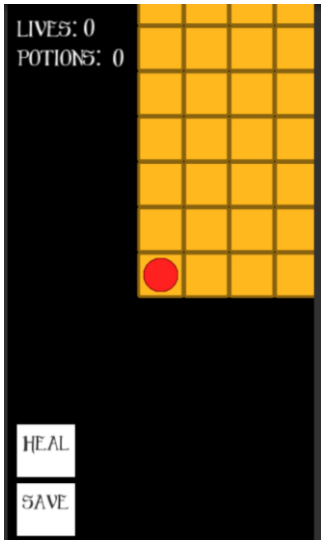
I decided to print the saved coordinates that were being assigned to the player to move it. I found that the saved coordinates were (0,0,0) which was why the player was spawning there (**figure 176, page 100**).

[Figure 176]



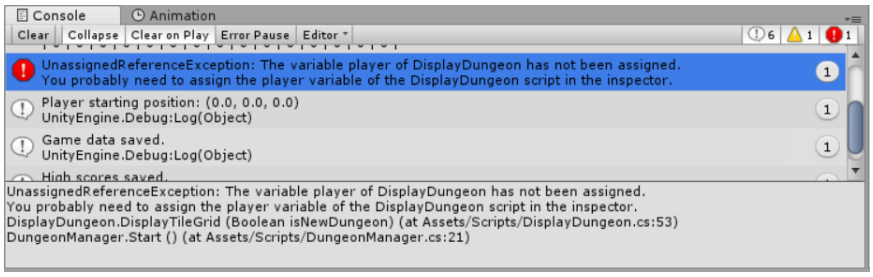
I added some temporary code to get rid of the current dungeon grid and generate a new one so that a new starting position would also be generated. When I ran my game, a new dungeon was generated with a new exit and starting coordinates (**figure 178, page 101**), but the player still stayed at (0,0,0) (**figure 177, page 101**) and I got an unassigned reference exception error (**figure 179, page 101**).

[Figure 177]



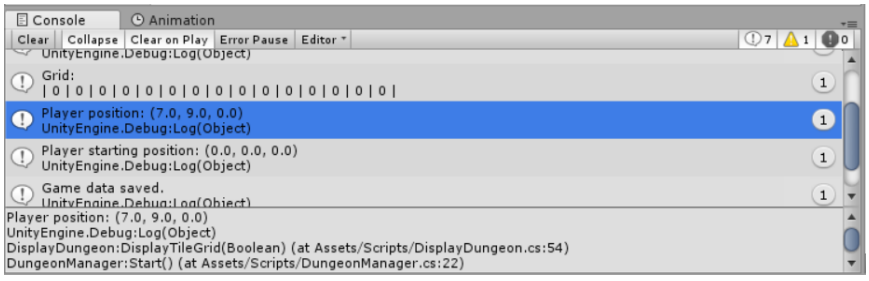
[Figure 178]

[Figure 179]



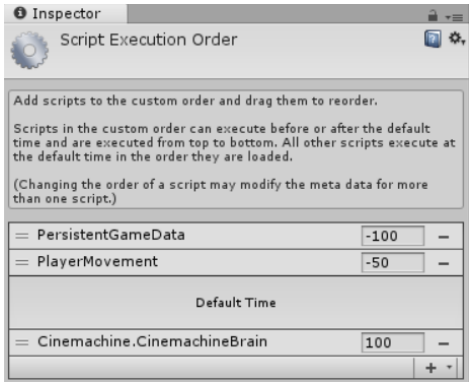
I realised I hadn't attached the player game object to the DisplayDungeon script in the Unity editor which was causing the problem. I attached it and tested again. This fixed the error I had before, but the player still spawned at (0,0,0). I decided to print the value of the player's position after it had been changed in the DisplayDungeon class (**figure 180, page 101**). When I ran the game, I found that the player had moved to the starting position, but I never saw it because immediately afterwards the player got put back to (0,0,0) by the PlayerMovement class. This was because when the player's position is decided, both classes set the player's position in the Start method, but the PlayerMovement class's Start method gets called after the DisplayDungeon class's Start method.

[Figure 180]



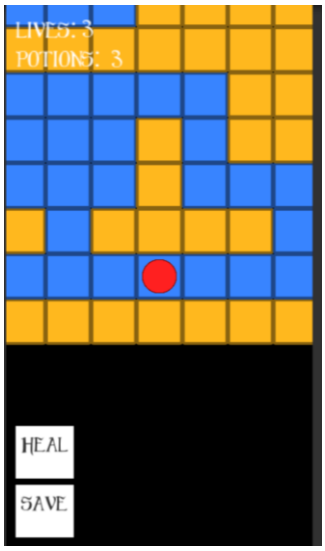
To fix this issue I changed the script execution order in Unity. I added the PersistentGameData to the top of the order because it should always be called first to load the saved data and because other classes depend on the data it loads. I then added PlayerMovement below it so that it would always be called before the DisplayDungeon and DungeonManager which are called in an Arbitrary order in the default time (**figure 181, page 102**).

[Figure 181]

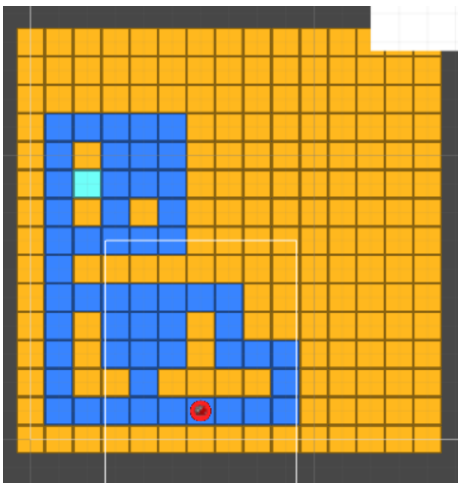


I tested the game again and this time the player was spawned on a floor tile on the map (**figures 182 & 183, page 102**).

[Figure 182]



[Figure 183]



I could also see from my printed coordinates in the console that the classes were being executed in the correct order (**figure 184, page 102**).

[Figure 184]

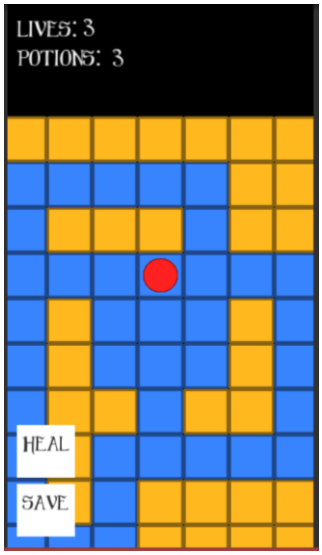
Since the test was successful I removed the debugging code I had added. After removed the debugging code I found that the player's position wasn't being saved so I added some code in the PlayerMovement class to save the position of the player (**figure 185, page 103**).

[Figure 185]

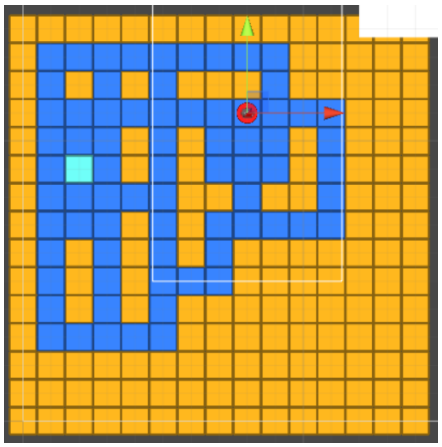
```
66    //Save the player's position before another scene is loaded.
67    void OnDisable () {
68        PersistentGameData.playerPosSave = gameObject.transform.position;
69    }
```

I tested it by generating a new dungeon and seeing if the starting position was saved when I reloaded my game (**figures 186 & 187, page 103**).

[Figure 186]

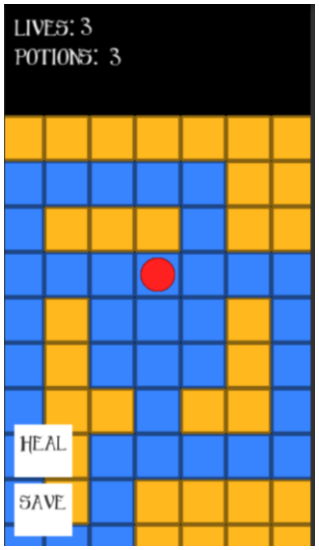


[Figure 187]

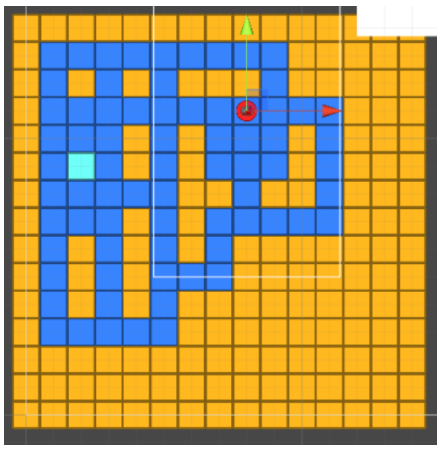


The test was successful as when I reloaded the game, the player spawned in the same position as before (**figures 186 & 187, page 103**).

[Figure 188]



[Figure 189]



Testing the pathfinding of the player character

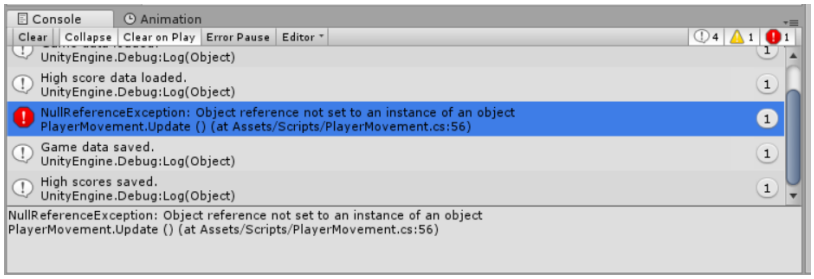
Since in my program, the player will only path-find to a tile being touched on a phone I added some debugging code that would make the player react to my mouse clicking a tile (**figure 190, page 104**).

[Figure 190]

```
32    //Checks every frame for the player's touch on screen, if the player selects a floor tile in the dungeon
33    //the player character will travel there.
34    void Update () {
35        if (characterMoving == true) {
36            AnimateWalk();
37        } else if (Input.touchCount == 1) {
38            Touch fingerPos = Input.GetTouch(0);
39            Vector2 tileTouchedPos = playerCam.ScreenToWorldPoint(fingerPos.position);
40            goal = Vector2Int.RoundToInt(tileTouchedPos);
41            if (goal.x > 0 && goal.x < gridLength && goal.y > 0 && goal.y < gridLength) {
42                if (dungeonGrid[goal.x, goal.y] != 0) {
43                    openNodes.Clear();
44                    visitedNodes.Clear();
45                    path.Clear();
46                    AStarPathfinding();
47                    characterMoving = true;
48                }
49            }
50        //Debugging condition for searching for mouse input.
51        } else if (Input.GetMouseButtonDown(0)) {
52            Vector2 tileTouchedPos = playerCam.ScreenToWorldPoint(Input.mousePosition);
53            goal = Vector2Int.RoundToInt(tileTouchedPos);
54            if (goal.x > 0 && goal.x < gridLength && goal.y > 0 && goal.y < gridLength) {
55                if (dungeonGrid[goal.x, goal.y] != 0) {
56                    openNodes.Clear();
57                    visitedNodes.Clear();
58                    path.Clear();
59                    AStarPathfinding();
60                    characterMoving = true;
61                }
```

I could then start testing the path finding. The first test I tried was picking a tile directly below the player as it is an easy path to find. However, when I clicked the tile, I got an error and nothing happened (**figure 191, page 104**).

[Figure 191]



I realised the error was because I was trying to clear the openNodes list in the PlayerMovement class but at that point I hadn't initialised the list yet. I edited my code to make sure I had initialised all the lists in the PlayerMovement class in the Start method (**figure 192, page 104**).

[Figure 192]

```
20    //Finds the camera, dungeon grid and gridLength.
21    void Start () {
22        playerCam = mainCamera.GetComponent<Camera>();
23        dungeonGrid = PersistentGameData.dungeonMapSave;
24        float totalElements = (float)PersistentGameData.dungeonMapSave.Length;
25        gridLength = Mathf.RoundToInt(Mathf.Sqrt(totalElements)) - 1;
26        characterMoving = false;
27        nextStepTime = 0f;
28        count = 0;
29        gameObject.transform.position = PersistentGameData.playerPosSave;
30        openNodes = new List<Node>();
31        visitedNodes = new List<Node>();
32        path = new List<Vector3Int>();
33    }
```

When I tried testing again the error was gone, but after I clicked a tile, my game froze and got stuck in an infinite loop which seemed to keep using up more space as I kept the game running. I tried to work out where the problem was by creating break points in my code and stepping through the values of Nodes at different parts of my program (**figure 193, page 105**). While stepping through the program I found an error I made with the conditions for adding a node to the openNodes list where if a duplicate was not found the node wasn't added to the list which is the opposite of what should happen. I also found another error that was causing the infinite loop which was that I had set the fCost of the starting Node to zero when I should have calculated it. This meant that this node had the lowest fCost so this node was the only one being tested.

[Figure 193]

```
89    //Pathfinds to the goal coordinates using the A* algorithm.
90    void AStarPathfinding () {
91        Vector2Int startCoords = Vector2Int.RoundToInt(gameObject.transform.position);
92        Debug.Log("StartCoords: " + startCoords.ToString());
93        Node startNode = new Node(startCoords, null, 0, 0);
94        int startFCost = CalculateFCost(startNode);
95        startNode.fCost = startFCost;
96        openNodes.Add(startNode);
97        Node currentNode = startNode;
98        //While the path has not reached the goal.
99        while (currentNode.coords != goal) {
100           Vector2Int[] neighbours = FindNeighbours(currentNode.coords);
101           foreach (Vector2Int n in neighbours) {
102               Debug.Log("Neighbour node: " + n.ToString());
103               if (n != Vector2Int.zero) {
104                   Node newNode = new Node(n, currentNode, 0, currentNode.gCost + 1);
105                   int FCost = CalculateFCost(newNode);
106                   newNode.fCost = FCost;
107                   Node duplicate = FindNodeInList(newNode, false);
108                   if (duplicate == null) {
109                       duplicate = FindNodeInList(newNode, true);
110                       if (duplicate != null && newNode.fCost < duplicate.fCost) {
111                           openNodes.Add(newNode);
112                       } else if (duplicate == null) {
113                           openNodes.Add(newNode);
114                       }
115                   }
116               }
117           }
```

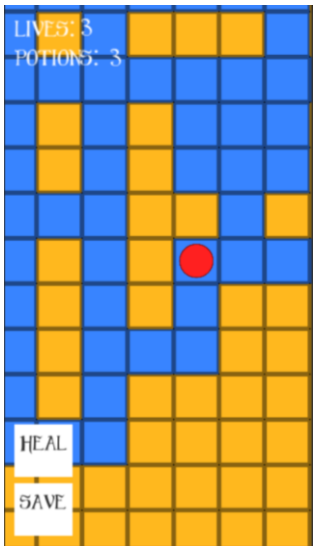
I stopped testing using the breakpoints and ran the game normally to see if my game did not crash anymore. It didn't crash but it still didn't work properly as I got an error and the player character didn't move (**figure 194, page 105**).

[Figure 194]

This error was because of my Node class which can't inherit from MonoBehaviour. I removed the inheritance and tested again. This time I received no errors and the player would move to the tile I clicked. I kept choosing positions to check my program worked in all situations. It worked almost all the time until I picked a position that was just across from the player but required going on a path around walls at which point my game froze (**figure 195, page 106**). I know my game froze at this

point because the tiles around the player would have worse fCost values than the current player position which would lead to an infinite loop.

[Figure 195]



I edited my code so that now the starting node had an fCost of the maximum value of an int variable (**figure 196, page 106**). This would make sure all other fCosts were lower than the starting node's fCost which shouldn't be considered when looking for the next node.

[Figure 196]

```
88    //Pathfinds to the goal coordinates using the A* algorithm.
89    void AStarPathfinding () {
90        Vector2Int startCoords = Vector2Int.RoundToInt(gameObject.transform.position);
91        Node startNode = new Node(startCoords, null, 0, 0);
92        startNode.fCost = int.MaxValue;
```

I tested my game by doing the same move again and checking my program didn't crash. It still did but this time it got stuck on the second node it checked. I realised this was because the heuristic I was using gave both tiles around the player the same hCost which gave them the same fCost when the tile below the player is closer to the goal than the tile to the right. My program kept picking the right tile and getting stuck on it. I decided to change the heuristic I was using from the Manhattan distance to the diagonal distance which I checked gave the bottom tile a smaller fCost. I changed my code to calculate this more accurate heuristic (**figure 197, page 106**).

[Figure 197]

```
153   //Calculates the fCost of a node using the diagonal distance heuristic.
154   int CalculateFCost (Node node) {
155       int xCoord = node.coords.x;
156       int yCoord = node.coords.y;
157       int hCost = Mathf.Max(Mathf.Abs(xCoord-goal.x), Mathf.Abs(yCoord-goal.y));
158       int fCost = node.gCost + hCost;
159       return fCost;
160   }
```

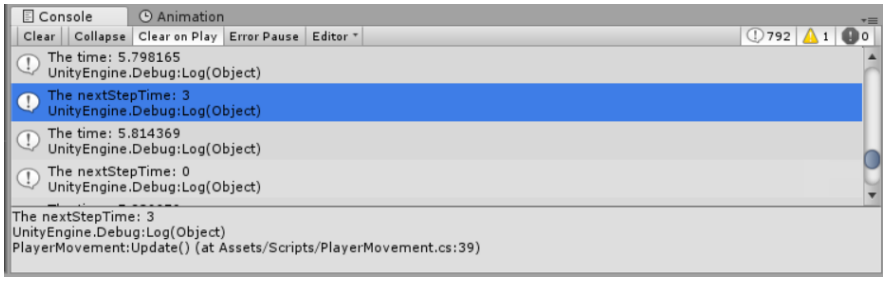
I tried the test again and this time the test worked however now, my program froze when I selected a tile that was diagonal to the current tile. So, I ended up with the same problem. I decided to edit my code again so that the current node had nothing to do with the next node being chosen. Instead of the just the start Node having a value of the maximum possible I changed it so that the current Node had the maximum value just before looking for the next node (**figure 198, page 107**).

[Figure 198]

```
113             visitedNodes.Add(currentNode);
114             openNodes.Remove(currentNode);
115             currentNode.fCost = int.MaxValue;
116             foreach (Node m in openNodes) {
117                 //Find the node with least fCost to be next current Node.
118                 if (m.fCost <= currentNode.fCost) {
119                     currentNode = m;
120                 }
121             }
```

I tested my code again and now the pathfinding worked properly with no freezing. I then wanted to change the speed of the movement because the character moved almost instantaneously through the dungeon. Changing the speed had no effect on walk so I printed the values of the time and the nextStepTime and found that the time kept increasing until it was always larger than the nextStepTime (**figure 199, page 107**).

[Figure 199]



I fixed my code by changing the Update (**figure 200, page 107**) and AnimateWalk (**figure 201, page 107**) methods to properly move the player at regular intervals.

[Figure 200]

```
35      //Checks every frame for the player's touch on screen, if the player selects a floor tile in the dungeon
36      //the player character will travel there.
37      void Update () {
38          if (characterMoving == true) {
39              tempTime += Time.deltaTime;
40              if (tempTime > movementSpeed) {
41                  AnimateWalk ();
42              }
```

[Figure 201]

```
77      //Animate the player character moving through the dungeon.
78      void AnimateWalk () {
79          tempTime = 0f;
80          if (count != path.Count) {
81              gameObject.transform.position = path[count];
82              count++;
83          } else {
84              PersistentGameData.playerPosSave = gameObject.transform.position;
85              characterMoving = false;
86              count = 0;
87          }
88      }
```

I tested my game and now the player moved slower at the walk speed I specified using the movement speed variable I had created. I tested a few times with different walk speeds until I found one I felt happy with. The speed ended up being walk a tile every 0.09 seconds.

The player reaching the end of the dungeon level

Now the player could move, I edited the PlayerMovement script to check if the player walks on the exit tile (**figure 202, page 108**). I checked to see if the player was at the same position as the exit. If the player reaches the exit, the level of the dungeon should increase, and a new dungeon should be generated after the scene reloads.

[Figure 202]

```
37    //Checks every frame for the player's touch on screen, if the player selects a floor tile in the dungeon
38    //the player character will travel there. If the player is at the exit a new dungeon level is loaded.
39    void Update () {
40        //If the player steps on the exit tile, the next level of the dungeon is loaded.
41        if (gameObject.transform.position == PersistentGameData.exitPosSave) {
42            PersistentGameData.dungeonLevel += 1;
43            PersistentGameData.dungeonMapSave = new int[1,1];
44            lvlManager.ReLoadLevel();
```

I tested this to make sure a new dungeon was generated.

Encountering a monster to battle

I added the chance for the player to run into a monster to the AnimateWalk method. I gave the method a 1 in 18 chance of running into a monster which means roughly every 18th tile stepped on ends up being a battle. I gave the PlayerMovement class access to the LevelManager class so that if the player does encounter a monster, the battle scene is loaded which is the next scene in the build settings (**figure 203, page 108**).

[Figure 203]

```
79    //Animate the player character moving through the dungeon and give them a 1 in 18 chance of
80    //encountering a monster.
81    void AnimateWalk () {
82        tempTime = 0f;
83        if (count != path.Count) {
84            gameObject.transform.position = path[count];
85            count++;
86            int chance = Random.Range(1, 19);
87            if (chance == 18) {
88                lvlManager.LoadNextAfterFade();
89            }
90        } else {
91            PersistentGameData.playerPosSave = gameObject.transform.position;
92            characterMoving = false;
93            count = 0;
94        }
95    }
```

I will test this when developing the Battle scene to make sure the battle scene is loaded with the correct data and that after battling the next scene will be this scene.

Allowing the use to zoom in and out by pinching the screen

To allow the user to zoom out and see more of the dungeon, I am creating a pinch zoom class to read the user's touch input. This class is based on the one created by Unity in their Mobile & Touch tutorials (https://unity3d.com/learn/tutorials/topics/mobile-touch/pinch-zoom?playlist=17138). I created a class called PinchZoom and attached it to the virtual Cinamachine camera in the scene (**figure 204, page 109**).

[Figure 204]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Cinemachine;

public class PinchZoom : MonoBehaviour {

    public float orthoZoomSpeed = 0.5f;

    private CinemachineVirtualCamera playerCam;

    //Find the camera component on this game object.
    void Start () {
        playerCam = GetComponent<CinemachineVirtualCamera>();
    }

    //Every frame, the class will check whether there are two fingers touching the screen of the device, it there are
    //it will do a couple of calculations to measure the difference between them and check if the user is trying to
    //zoom out or in and the camera size will change depending on the touch inputs.
    void Update () {
        if (Input.touchCount == 2) {
            Touch touchZero = Input.GetTouch(0);
            Touch touchOne = Input.GetTouch(1);
            Vector2 touchZeroPrevPos = touchZero.position - touchZero.deltaPosition;
            Vector2 touchOnePrevPos = touchOne.position - touchOne.deltaPosition;
            float prevTouchDeltaMag = (touchZeroPrevPos - touchOnePrevPos).magnitude;
            float touchDeltaMag = (touchZero.position - touchOne.position).magnitude;
            float deltaMagnitudeDif = prevTouchDeltaMag - touchDeltaMag;
            playerCam.m_Lens.OrthographicSize += deltaMagnitudeDif * orthoZoomSpeed;
            playerCam.m_Lens.OrthographicSize = Mathf.Max(playerCam.m_Lens.OrthographicSize, 2.5f);
        }
    }
}
```

I can only test this when I use my phone to test so I will leave testing this class until I have developed all of my game.

Scene 4 Review

A lot was done to this scene as this was where I created half of the actual game's systems. I have now met success criteria 1.4, 1.5, 2.7, 2.8, 2.9 and part of 1.7, 2.12 and 2.13. The dungeons are randomly generated and are displayed to the user as well as the player character which can move around the dungeon. At the moment, I am just using test images as the sprites of each object but when the game is almost completed, I will replace the test sprites with actual images that fit the game's fantasy setting. Developing the dungeon generation algorithm took the most steps and testing as is very complex and I learned new ways of coding and testing throughout the process such as creating a predicate to search through lists easier. I also learned how to use breakpoints and how to expose variables in MonoDevelop to help me debug my programs. Programming the A* pathfinding took a long time as well because I encountered a lot of bugs during its development and I was worried that the code to perform the pathfinding method would take too long to process by the computer, so it would take a while before the game showed the player moving but when I was testing, the character started moving almost instantaneously after I had selected a tile to walk to. The pathfinding should also have been made quicker by the dungeons being generated without dead ends, so my program doesn't have to do a lot of backtracking. In my design, I didn't write full pseudocode algorithms for the classes on this scene because I knew they would all work together closely and if I made changes to one it would affect the others a lot and also because they would be very long to write and need a lot of testing to make sure they worked correctly.

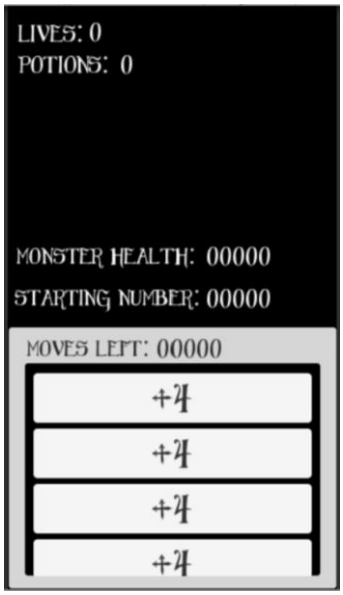
Scene 5: Battle

Creating the user interface for the battle scene

I started creating the UI for this scene by adding labels for the information that is displayed to the user and display boxes to display them. I then added a panelled section on the bottom half of the screen where the button menu would be displayed. I created the first menu with the fight and heal buttons (**figure 205, page 110**). The heal button should act the same way as the heal button from the Dungeon scene. The fight button will display the second menu (**figure 206, page 110**) which shows the mathematical operation buttons and gives the player the chance to defeat the monster. I created a scrollable area to the second menu because there could be a list of many buttons that go off the screen. While doing this I created a new button prefab specifically for the buttons that display operations.

[Figure 205]

[Figure 206]



Developing the OptionsMenu and GenerateMathQuestion classes

I then started creating the class called OptionsMenu from my design section which was almost identical to the DungeonManager class on the previous level that handled displaying the values of the number of lives, potions, monster health, moves left and the starting number. I created references to the text boxes that displayed these variables and added references to them. I then attached the text box game objects to the class in Unity. I then added a heal method for the heal button to call on menu 1. I copied the code to do this from the Dungeon Manager but when I was copying I realised I had made an error because the player can only heal themselves if they have potions. If they don't have any potions it won't work. So, I added a condition to healing to the methods from both classes to check that the player has potions to use (**figure 207, page 110**).

[Figure 207]

```
23    //When the player presses the heal button, one potion is used to give the player full health.
24    public void Heal () {
25        if (PersistentGameData.potions > 0) {
26            PersistentGameData.potions -= 1;
27            PersistentGameData.playerLivesSave = 3;
28            lives.text = PersistentGameData.playerLivesSave.ToString();
29            potions.text = PersistentGameData.potions.ToString();
30        }
31    }
```

I then gave the method access to the MenuOne and MenuTwo game objects so that this class could enable and disable the two menus. I created the Fight method that the fight button on MenuOne would call to disable MenuOne and enable MenuTwo. In Unity I attached the Heal and Fight buttons to their respective method from the OptionsMenu class. Before continuing to develop this class, I needed to create the GenerateMathQuestion class. I attached this class to the same game object as the OptionsMenu class. I created this class according to my designs from the design section. While creating this class I improved my original pseudocode designs to make the class more efficient. I also decided to convert the integer array of operator codes in the PersistentGameData (**figure 208, page 111**) class to a list so that the size of the list would always be the number of operators the user picked. I changed the references to the operatorCodesArray in all the classes that used it to now work with the data as a list which were the GameData class (**figure 209, page 111**) and the CreateNewGame class (**figure 210, page 111**).

[Figure 208]

```
6  public class PersistentGameData : MonoBehaviour {
7
8      public static int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
9      public static string playerName;
10     public static Vector3 playerPosSave, exitPosSave;
11     public static List<int> operatorCodesArray;
12     public static int[,] dungeonMapSave;
13
14     //When this object is instantiated, this class will load the saved data from the file or create a set of new data
15     //if there is no file and this is a new game.
16     void Awake () {
17         bool dataLoaded = LoadGameData();
18         if (!dataLoaded) {
19             dungeonMapSave = new int[1,1];
20             operatorCodesArray = new List<int>();
21             playerPosSave = new Vector3(0,0,0);
22             exitPosSave = new Vector3(0,0,0);
23             dungeonLevel = 1;
24             playerLivesSave = 3;
25             potions = 3;
26             rangeMin = 0;
27             rangeMax = 0;
28             maxMoves = 0;
29             playerName = "";
30         }
31     }
```

[Figure 209]

```
84 //This is a separate private class that inherits from SaveGame and is used by PersistentGameData to store the game data
85 //in a serialized permentant file.
86 [Serializable]
87 class GameData : SaveGame {
88
89     public int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
90     public string playerName;
91     public float playerPosSaveX, playerPosSaveY, playerPosSaveZ;
92     public float exitPosSaveX, exitPosSaveY, exitPosSaveZ;
93     public List<int> operatorCodesArray;
94     public int[,] dungeonMapSave;
95
96     //The vector of the player's and exit's position is saved separately as three floats of each separate coordinate
97     //because Vector3 is not a standard data type.
98 }
```

[Figure 210]

```
96     //The operatorCodesArray stores the different operations the player wants to use, 1 is addition, 2 is
97     //subtraction, 3 is multiplication and 4 is division.The different toggles corespond to different
98     //operators.
99     void GetOperatorCodes () {
100        PersistentGameData.operatorCodesArray = new List<int>();
101        validSelection = false;
102        for (int i = 0; i <= 3; i++) {
103            if (myToggles[i].isOn == true) {
104                PersistentGameData.operatorCodesArray.Add(i + 1);
105                validSelection = true;
106            }
107        }
108    }
```

I then continued to develop the GenerateMathQuestion class until it was complete (**figure 211, page 112**).

[Figure 211]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GenerateMathQuestion : MonoBehaviour {

    private int numberOfMoves, startNumber, targetNumber;
    private List<int[]> answerSequence;

    //Initialise the answerSequence list.
    void Start () {
        answerSequence = new List<int[]>();
    }

    //This function is called by the options menu class to get the number of
    //moves, the target number and the starting number of the question this
    //class generates.
    public int[] GetMathsQuestion () {
        GenerateQuestion();
        int[] results = new int[3] {numberOfMoves, startNumber, targetNumber};
        return results;
    }

    //This function is called by the options class menu to get the answer
    //sequence of the question generated.
    public List<int[]> GetAnswerSequence () {
        return answerSequence;
    }
    //This procedure performs all the required steps to generate a maths
    //question such as generating a target number and the number of moves
    //the question will take to solve. It generates each move and adds
    //them to a sequence to be passed to the options class.
    void GenerateQuestion () {
        targetNumber = Random.Range(PersistentGameData.rangeMin, PersistentGameData.rangeMax + 1);
        numberOfMoves = Random.Range(1, PersistentGameData.maxMoves +1);
        startNumber = targetNumber;
        for (int moveNumber = 0; moveNumber < numberOfMoves; moveNumber++) {
            int[] moveGenerated = GenerateMove(startNumber);
            answerSequence.Add(moveGenerated);
        }
    }
    //This function generates a move by generating an operator and an integer
    //that the operation will be performed on. The start number is modified by
    //performing the opposite operator on it. When the operator chosen is
    //multiplication, the operator number can only be a factor of the current
    //target to keep all the sums using integer values.
    int[] GenerateMove (int currentTarget) {
        int operatorsNumber = Random.Range(0, currentTarget+1);
        int operation = Random.Range(0, PersistentGameData.operatorCodesArray.Count);
        int thisOperator = PersistentGameData.operatorCodesArray[operation];
        switch (thisOperator) {
            case 1:
                startNumber = targetNumber - currentTarget;
                break;
            case 2:
                startNumber = targetNumber + currentTarget;
                break;
            case 4:
                startNumber = targetNumber * currentTarget;
                break;
            default:
                while ((operatorsNumber % currentTarget) != 0) {
                    operatorsNumber = Random.Range(0, currentTarget);
                }
                startNumber = targetNumber / currentTarget;
                break;
        }
        int[] move = new int[] {thisOperator, operatorsNumber};
        return move;
    }
}
```

I then went back to developing the OptionsMenu class now that I had the data from the GenerateMathQuestion. I first added the code to access the GenerateMathQuestion class and get the question data and answer data from it. I added code to display the question data using the display boxes from the user interface. The answer data needs to be shown on individual buttons. Each button should have a label of what operation it is. When any button is pressed it should decrease the moves left variable and there should be an array keeping track of the order in which buttons were pressed. When I was trying find out how to print the labels for each button I decided to change the operatorCodesArray again as there was no use for representing each operation as a number code, instead each operator code would be a char variable that would be the operation's mathematical symbol. I edited all the classes that used the operatorCodesArray again which were the PersistentGameData class (**figure 213, page 113**), the GameData class (**figure 214, page 114**), the CreateNewGame class (**figure 212, page 113**) and the GenerateMathQuestion class (**figure 215, page 114**).

[Figure 212]

```
96      //The operatorCodesArray stores the different operations the player wants to use, 1 is addition, 2 is
97      //subtraction, 3 is multiplication and 4 is division.The different toggles corespond to different
98      //operators.
99      void GetOperatorCodes () {
100         PersistentGameData.operatorCodesArray = new List<char>();
101         validSelection = false;
102         for (int i = 0; i <= 3; i++) {
103             validSelection = true;
104             if (myToggles[i].isOn == true) {
105                 switch (i) {
106                     case 1:
107                         PersistentGameData.operatorCodesArray.Add('+');
108                         break;
109                     case 2:
110                         PersistentGameData.operatorCodesArray.Add('-');
111                         break;
112                     case 3:
113                         PersistentGameData.operatorCodesArray.Add('×');
114                         break;
115                     default:
116                         PersistentGameData.operatorCodesArray.Add('÷');
117                         break;
118                 }
119             }
120         }
121     }
122 }
```

[Figure 213]

```
6  public class PersistentGameData : MonoBehaviour {
7
8      public static int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
9      public static string playerName;
10     public static Vector3 playerPosSave, exitPosSave;
11     public static List<char> operatorCodesArray;
12     public static int[,] dungeonMapSave;
13
14     //When this object is instantiated, this class will load the saved data from the file or
15     //create a set of new data if there is no file and this is a new game.
16     void Awake () {
17         bool dataLoaded = LoadGameData();
18         if (!dataLoaded) {
19             dungeonMapSave = new int[1,1];
20             operatorCodesArray = new List<char>();
21             playerPosSave = new Vector3(0,0,0);
22             exitPosSave = new Vector3(0,0,0);
23             dungeonLevel = 1;
24             playerLivesSave = 3;
25             potions = 3;
26             rangeMin = 0;
27             rangeMax = 0;
28             maxMoves = 0;
29             playerName = "";
30         }
31     }
```

[Figure 214]

```
//This is a separate private class that inherits from SaveGame and is used by PersistentGameData to store the game data
//in a serialized permentant file.
[Serializable]
class GameData : SaveGame {

    public int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
    public string playerName;
    public float playerPosSaveX, playerPosSaveY, playerPosSaveZ;
    public float exitPosSaveX, exitPosSaveY, exitPosSaveZ;
    public List<char> operatorCodesArray;
    public int[,] dungeonMapSave;

    //The vector of the player's and exit's position is saved separately as three floats of each separate coordinate
    //because Vector3 is not a standard data type.
}
```

[Figure 215]

```
    //This function generates a move by generating an operator and an integer
    //that the operation will be performed on. The start number is modified by
    //performing the opposite operator on it. When the operator chosen is
    //multiplication, the operator number can only be a factor of the current
    //target to keep all the sums using integer values.
    int[] GenerateMove (int currentTarget) {
        int operatorsNumber = Random.Range(0, currentTarget+1);
        int operation = Random.Range(0, PersistentGameData.operatorCodesArray.Count);
        char thisOperator = PersistentGameData.operatorCodesArray[operation];
        switch (thisOperator) {
            case '+':
                startNumber = targetNumber - currentTarget;
                break;
            case '-':
                startNumber = targetNumber + currentTarget;
                break;
            case '÷':
                startNumber = targetNumber * currentTarget;
                break;
            default:
                while ((operatorsNumber % currentTarget) != 0) {
                    operatorsNumber = Random.Range(0, currentTarget);
                }
                startNumber = targetNumber / currentTarget;
                break;
        }
        int operatorUnicode = (int)System.Char.GetNumericValue(thisOperator);
        int[] move = new int[] {operatorUnicode, operatorsNumber};
        return move;
    }
}
```

After making the buttons, I created the conditions that would be continually checked to end the battle. If the player uses all their moves, their answers must be checked. If they are correct they defeat the monster and continue back to the dungeon scene. If they are not correct, the number of moves is reset, and the player loses a life. If the player loses all their lives, they die and must start a new game. To tell the player what was happening I created a third menu game object to display messages to the player while battling. To make sure that the save game was deleted, I edited the PersistentGameData class to only save the game data if a variable called saveTheGame was true (**figure 216, page 115**).

[Figure 216]

```
public class PersistentGameData : MonoBehaviour {

    public static int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
    public static string playerName;
    public static Vector3 playerPosSave, exitPosSave;
    public static List<char> operatorCodesArray;
    public static int[,] dungeonMapSave;
    public static bool saveTheGame;

    //When this object is instantiated, this class will load the saved data from the file or
    //create a set of new data if there is no file and this is a new game.
    void Awake () {
        saveTheGame = true;
        bool dataLoaded = LoadGameData();
        if (!dataLoaded) {
            dungeonMapSave = new int[1,1];
            operatorCodesArray = new List<char>();
            playerPosSave = new Vector3(0,0,0);
            exitPosSave = new Vector3(0,0,0);
            dungeonLevel = 1;
            playerLivesSave = 3;
            potions = 3;
            rangeMin = 0;
            rangeMax = 0;
            maxMoves = 0;
            playerName = "";
        }
    }
    //Before this object is destroyed, save the game data to the file.
    void OnDisable () {
        if (saveTheGame == true) {
            SaveGameData();
        }
    }
```

I finished the options class by adding references to the PersistentHighScores class to save the high scores (**Figure 217, page 115**).

[Figure 217]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class OptionsMenu : MonoBehaviour {

    public GameObject livesDisplay, potionsDisplay, monsterHealthDisplay, startingNumberDisplay, movesLeftDisplay;
    public GameObject menu1, menu2, menu3, buttonGrid;
    public GameObject operatorButtonPrefab;

    private LevelManager lvlManager;
    private Text lives, potions, monsterHealth, startingNumber, movesLeft, messageDisplay;
    private GenerateMathQuestion genMathQ;
    private List<int[]> answerOperations;
    private List<int> playerAnswers;
    private int[] mathsQuestion;
    private int movesLeftValue, state;
    //Initialise all game objects to display data to the user and generate a maths question.
    void Start () {
        lvlManager = GameObject.FindObjectOfType<LevelManager>();
        genMathQ = GetComponent<GenerateMathQuestion>();
        messageDisplay = menu3.GetComponentInChildren<Text>();
        lives = livesDisplay.GetComponent<Text>();
        potions = potionsDisplay.GetComponent<Text>();
        monsterHealth = monsterHealthDisplay.GetComponent<Text>();
        startingNumber = startingNumberDisplay.GetComponent<Text>();
        movesLeft = movesLeftDisplay.GetComponent<Text>();
        lives.text = PersistentGameData.playerLivesSave.ToString();
        potions.text = PersistentGameData.potions.ToString();
        mathsQuestion = genMathQ.GetMathsQuestion();
        monsterHealth.text = mathsQuestion[2].ToString();
        startingNumber.text = mathsQuestion[1].ToString();
        movesLeftValue = mathsQuestion[0];
        movesLeft.text = movesLeftValue.ToString();
        answerOperations = genMathQ.GetAnswerSequence();
        answerOperations.Reverse();
        CreateButtons();
    }
```

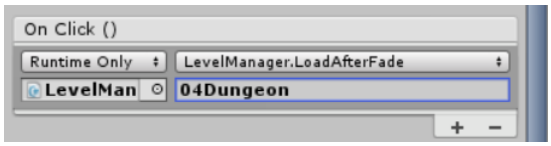
```
42    //Check if the player has answered the question or died.
43    void Update () {
44        if (movesLeftValue == 0) {
45            bool correct = true;
46            for (int i = 0; i < playerAnswers.Count; i++) {
47                if (playerAnswers[i] != i) {
48                    correct = false;
49                }
50            }
51            if (correct) {
52                //Player has won battle.
53                PersistentGameData.potions += 1;
54                PersistentHighScores.enemiesDefeated += 1;
55                menu3.SetActive(true);
56                messageDisplay.text = "You have won! You received a potion as a reward.";
57                state = 1;
58                Invoke("DealWithOutcome", 2);
59            } else {
60                //Player was not correct.
61                PersistentGameData.playerLivesSave -= 1;
62                lives.text = PersistentGameData.playerLivesSave.ToString();
63                movesLeftValue = mathsQuestion[0];
64                menu3.SetActive(true);
65                messageDisplay.text = "You were not correct. The monster attacked and you lost a life.";
66                state = 2;
67                Invoke("DealWithOutcome", 2);
68            }
69        } else if (PersistentGameData.playerLivesSave == 0) {
70            //Player is dead.
71            if (PersistentGameData.dungeonLevel > PersistentHighScores.maxFloorsCleared) {
72                PersistentHighScores.maxFloorsCleared = PersistentGameData.dungeonLevel;
73            }
74            menu3.SetActive(true);
75            messageDisplay.text = "You are dead. Game Over.";
76            state = 3;
77            Invoke("DealWithOutcome", 2);
78        }
79    }
80
81    //The outcome of the player answering a question or dying.
82    void DealWithOutcome () {
83        switch (state) {
84            case 1:
85                lvlManager.LoadAfterFade("04Dungeon");
86                break;
87            case 2:
88                menu3.SetActive(false);
89                menu2.SetActive(false);
90                menu1.SetActive(true);
91                break;
92            case 3:
93                SaveGameSystem.DeleteSaveGame("SavedGameData");
94                PersistentGameData.saveTheGame = false;
95                lvlManager.LoadAfterFade("01StartMenu");
96                break;
97        }
98    }
100    //Creates the list of buttons for the user to enter their answer.
101    void CreateButtons () {
102        int buttonNumber = answerOperations.Count;
103        List<int> numbers = new List<int>();
104        playerAnswers = new List<int>();
105        for (int j = 0; j < buttonNumber; j++) {
106            numbers.Add(j);
107        }
108        for (int i = 0; i < buttonNumber; i++) {
109            GameObject operationButton = Object.Instantiate(operatorButtonPrefab, buttonGrid.transform) as GameObject;
110            Text operationButtonText = operationButton.GetComponentInChildren<Text>();
111            Button operationButtonFunction = operationButton.GetComponent<Button>();
112            int randomPosition = Random.Range(0, numbers.Count);
113            numbers.Remove(randomPosition);
114            string buttonText = System.Convert.ToChar(answerOperations[numbers[randomPosition]][0]).ToString();
115            buttonText = System.String.Concat(buttonText, answerOperations[numbers[randomPosition]][1].ToString());
116            operationButtonText.text = buttonText;
117            operationButtonFunction.onClick.AddListener(delegate {WhenButtonClicked(numbers[randomPosition]); });
118        }
119    }
120
121    //All buttons have this function to record the player's answer.
122    void WhenButtonClicked (int buttonNumber) {
123        movesLeftValue -= 1;
124        playerAnswers.Add(buttonNumber);
125    }
```

```
127     //When the player presses the heal button, one potion is used to give the player full health.
128     public void Heal () {
129         if (PersistentGameData.potions > 0) {
130             PersistentGameData.potions -= 1;
131             PersistentGameData.playerLivesSave = 3;
132             lives.text = PersistentGameData.playerLivesSave.ToString();
133             potions.text = PersistentGameData.potions.ToString();
134         }
135     }
136 
137     //Called by the fight button to switch the visible menus.
138     public void Fight () {
139         menu2.SetActive(true);
140         menu1.SetActive(false);
141     }
142 }
```

To test this class, I decided to test my game from the start since this class depends on the settings chosen in the NewGame scene.

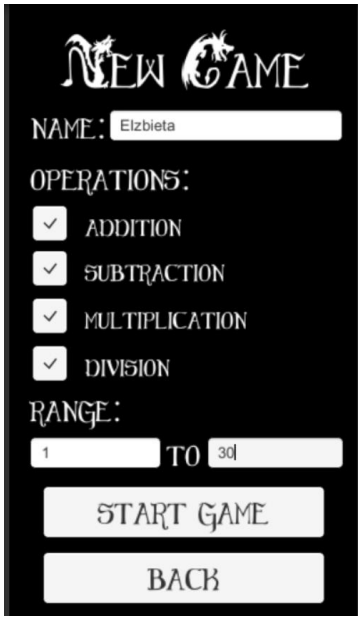
Before I could test my game, I found I hadn't added any functionality to the continue game button on the Start Menu scene, so I added the functionality to it of loading the Dungeon scene (**figure 218, page 117**).

[Figure 218]



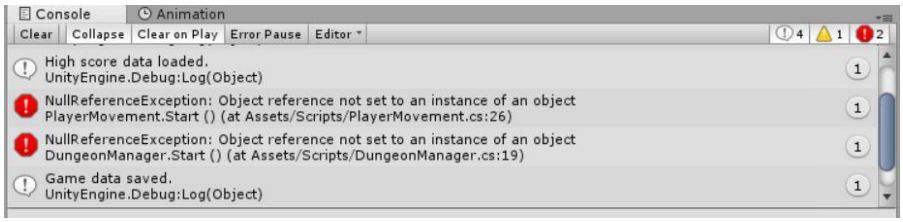
I then continued to test and create a new game. This is the test data I used (**figure 219, page 117**).

[Figure 219]



When I clicked the Start Game button, the Dungeon scene was loaded but the dungeon on it was not and I got two error messages (**figure 220, page 118**).

[Figure 220]



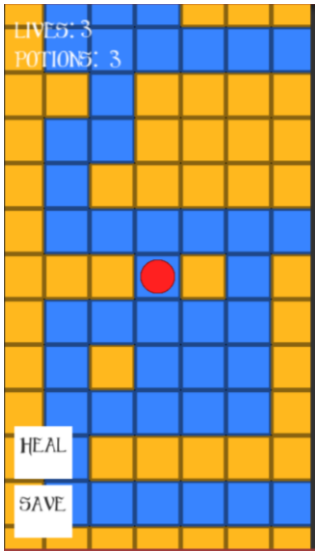
To try and fix this, I edited the CreateNewGame class so that it would reset all the data that is saved when a new game is created (**figure 221, page 118**).

[Figure 221]

```
24    //Called when the player presses the Start Game button and reads what the player has entered as well as
25    //creating new data to be saved.
26    public void CreateGame () {
27        bool correctInput = ValidateInputs();
28        GetOperatorCodes();
29        if (validSelection == false) {
30            OpenDialogBox("Please pick at least one operator to use.");
31        } else if (correctInput == true) {
32            PersistentGameData.maxMoves = CalculateMaxMoves (PersistentGameData.dungeonLevel);
33            PersistentGameData.dungeonMapSave = new int[1,1];
34            PersistentGameData.playerPosSave = new Vector3(0,0,0);
35            PersistentGameData.exitPosSave = new Vector3(0,0,0);
36            PersistentGameData.dungeonLevel = 1;
37            PersistentGameData.playerLivesSave = 3;
38            PersistentGameData.potions = 3;
39            lvlManager.LoadNextAfterFade();
40        }
41    }
```

This fixed the error as now the Dungeon scene was loaded correctly with no errors (**figure 222, page 118**).

[Figure 222]



I tried clicking to move my character around the dungeon, but it didn't move. I realised that the player wasn't moving because the PlayerMovement class wasn't loading the newly created dungeon grid as the grid was generated after the class had loaded the initial dungeon. This meant the class had no array of nodes to look through when pathfinding. I added a method called GetNewDungeon (**figure 223, page 119**) to this class to get the new dungeon grid after a new one had been called to be generated by the DisplayDungeon class (**figure 224, page 119**).

[Figure 223]

```
86      //Called by the DisplayDungeon class to get the newly generated dungeon.
87      public void GetNewDungeon () {
88          dungeonGrid = PersistentGameData.dungeonMapSave;
89          float totalElements = (float)PersistentGameData.dungeonMapSave.Length;
90          gridLength = Mathf.RoundToInt(Mathf.Sqrt(totalElements)) - 1;
91      }
```

[Figure 224]

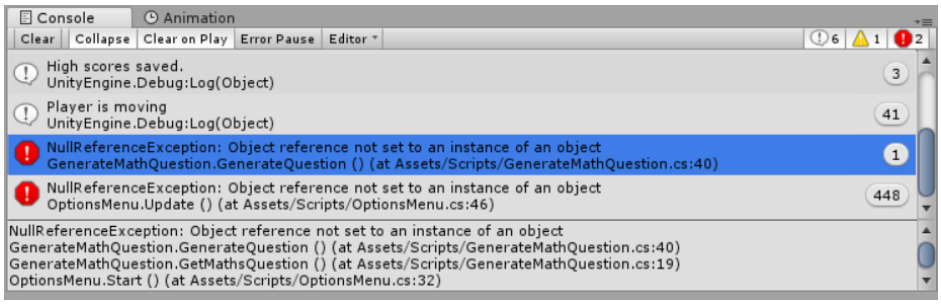
```
42          //Picks the position of the dungeon exit and the player's starting position if a new dungeon is generated,
43          //otherwise the exit is placed at the coordinates loaded from memory.
44          if (isNewDungeon == true) {
45              int exitPosition;
46              exitPosition = Random.Range(0, newTile);
47              exitMap.SetTile(floorTileCoords[exitPosition], exit);
48              PersistentGameData.exitPosSave = (Vector3)floorTileCoords[exitPosition];
49              int entrancePosition;
50              do {
51                  entrancePosition = Random.Range(0, newTile);
52                  startingPosition = floorTileCoords[entrancePosition];
53                  player.transform.position = startingPosition;
54                  PersistentGameData.playerPosSave = startingPosition;
55              } while (exitPosition == entrancePosition);
56              PlayerMovement movement = player.GetComponent<PlayerMovement>();
57              movement.GetNewDungeon();
58          } else {
59              Vector3Int exitCoords;
60              exitCoords = Vector3Int.RoundToInt(PersistentGameData.exitPosSave);
61              exitMap.SetTile(exitCoords, exit);
62          }
63      }
```

I tested my whole game again and now the player could move. I then checked that the save button worked by pressing it to return to the start menu. On the start menu I pressed the continue game button to return back to my saved game. The game was loaded exactly how I expected it to, so it worked. To test that the heal button worked I first wanted to get into a battle and lose some health so that I could check the player gets their full health back.

When I encountered a battle, none of the question data was displayed (**figure 225, page 119**) and I received two errors in the console (**figure 226, page 120**).

[Figure 225]

[Figure 226]



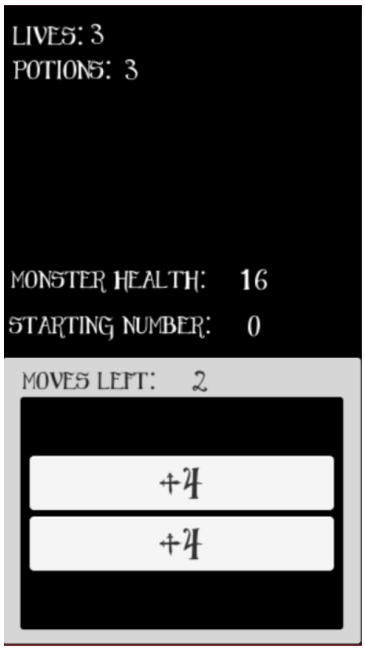
I realised it was because in the GenerateMathQuestion, the answerSequence list was being accessed before it was instantiated because the OptionsMenu class which calls the GenerateMathQuestion doesn't allow this class to call it's Start method where the list is instantiated. I changed the code in this class so that now the list is instantiated in the method that uses it and there is no longer any Start method (**figure 227, page 120**).

[Figure 227]

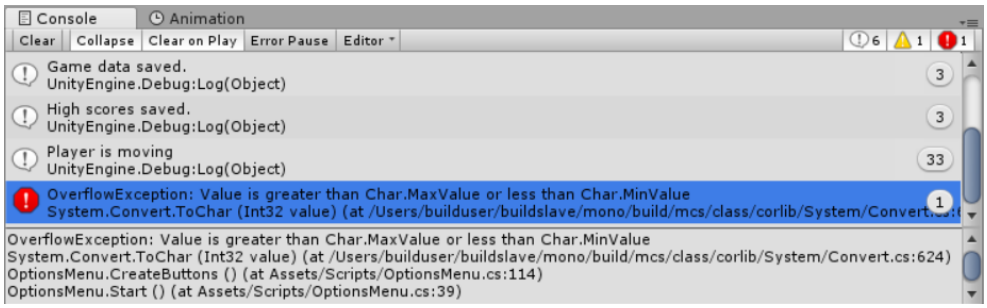
```
25    //This procedure performs all the required steps to generate a maths
26    //question such as generating a target number and the number of moves
27    //the question will take to solve. It generates each move and adds
28    //them to a sequence to be passed to the options class.
29    void GenerateQuestion () {
30        answerSequence = new List<int[]>();
31        targetNumber = Random.Range(PersistentGameData.rangeMin, PersistentGameData.rangeMax + 1);
32        numberOfMoves = Random.Range(1, PersistentGameData.maxMoves +1);
33        startNumber = targetNumber;
34        for (int moveNumber = 0; moveNumber < numberOfMoves; moveNumber++) {
35            int[] moveGenerated = GenerateMove(startNumber);
36            answerSequence.Add(moveGenerated);
37        }
38    }
```

I tested my game again and the data was now displayed but the buttons weren't working correctly as they were all displaying as the default button (**figure 228**, **page 120**) and I received an overflow error from the console (**figure 229**, **page 121**).

[Figure 228]



[Figure 229]



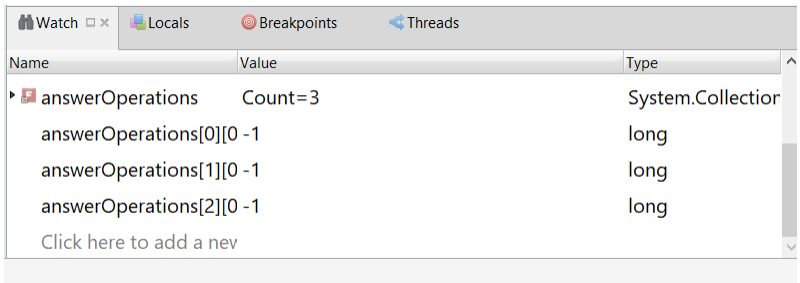
This error happened when converting the integer representing the Unicode character to a char variable. To fix this I decided to change the data type of the Unicode value to a long variable since they can store larger numbers (**figure 230, page 121**).

[Figure 230]

```
45    long[] GenerateMove (int currentTarget) {
46        int operatorsNumber = Random.Range(0, currentTarget+1);
47        int operation = Random.Range(0, PersistentGameData.operatorCodesArray.Count);
48        char thisOperator = PersistentGameData.operatorCodesArray[operation];
49        switch (thisOperator) {
50            case '+':
51                startNumber = targetNumber - currentTarget;
52                break;
53            case '-':
54                startNumber = targetNumber + currentTarget;
55                break;
56            case '÷':
57                startNumber = targetNumber * currentTarget;
58                break;
59            default:
60                while ((operatorsNumber % currentTarget) != 0) {
61                    operatorsNumber = Random.Range(0, currentTarget);
62                }
63                startNumber = targetNumber / currentTarget;
64                break;
65        }
66        long operatorUnicode = (long)System.Char.GetNumericValue(thisOperator);
67        long operatorsNumberLong = System.Convert.ToInt64(operatorsNumber);
68        long[] move = new long[] {operatorUnicode, operatorsNumberLong};
69        return move;
70    }
```

I tested again but I got an out of range exception from trying to convert the Unicode values. I put breakpoints in my program to check what the values where and why they were out of range. I found that in the GenerateMathQuestion class the conversion from a char to a Unicode character code had been unsuccessful as each conversion was showing -1 as the code which means the conversion was unsuccessful (**figure 231, page 121**).

[Figure 231]



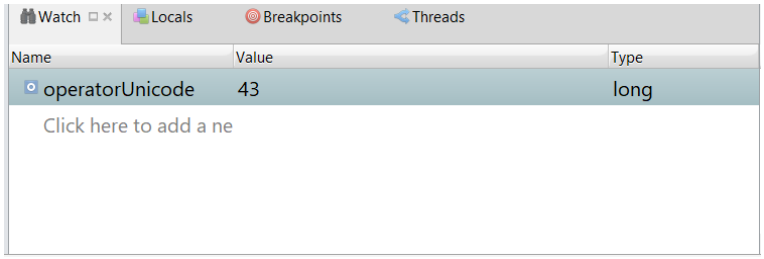
I realised this was because I was using the wrong method to find the character's Unicode as the method I was using just converted the value of the char if it was a number to an int which none of my characters were. I changed my code to use the correct way of converting from an int to a char which was implicitly (**figure 232, page 122**).

[Figure 232]

```
45      long[] GenerateMove (int currentTarget) {
46          int operatorsNumber = Random.Range(1, currentTarget+1);
47          int operation = Random.Range(0, PersistentGameData.operatorCodesArray.Count);
48          char thisOperator = PersistentGameData.operatorCodesArray[operation];
49          switch (thisOperator) {
50              case '+':
51                  startNumber = targetNumber - currentTarget;
52                  break;
53              case '-':
54                  startNumber = targetNumber + currentTarget;
55                  break;
56              case '÷':
57                  startNumber = targetNumber * currentTarget;
58                  break;
59              default:
60                  while ((operatorsNumber % currentTarget) != 0) {
61                      operatorsNumber = Random.Range(0, currentTarget);
62                  }
63                  startNumber = targetNumber / currentTarget;
64                  break;
65          }
66          long operatorUnicode = thisOperator;
67          long operatorsNumberLong = System.Convert.ToInt64(operatorsNumber);
68          long[] move = new long[] {operatorUnicode, operatorsNumberLong};
69          return move;
70      }
```

Now when I ran my code and exposed the value of the operatorUnicode value which gave the actual Unicode numbers (**figure 233, page 122**) which I checked were correct by searching them online.

[Figure 233]



I then realised that the Unicode values were not as big as I thought they were so the overflow exception error from before must have come from another part of my program or because of my incorrect code from before so I changed the long variable types back to integer types to make sure my game did not use unnecessary memory to store variables.

Before I continued debugging my game, I wanted to change the way my GenerateMathQuestion class generates moves as currently when the class is generating a multiplication move, it works backwards to find a number the current target can be divided by and end up with an integer value because my game only allows players to do integer mental maths. Currently my program tries to randomly come up with these integers to divide by, but this is not guaranteed to happen and is inefficient especially if the target number is a prime number, so I had the idea to instead find the prime factors of the target number and multiply a random combinations of these to find a number the target number could be divided by and still be an integer. I found an efficient algorithm to do this on the website https://www.geeksforgeeks.org/print-all-prime-factors-of-a-given-number/. I decided to add a new method to my GenerateMathQuestion class to perform this algorithm called

FindIntegerDevisor (**figure 235, page 123**) which would be called from the GenerateMove method (**figure 234, page 123**).

[Figure 234]

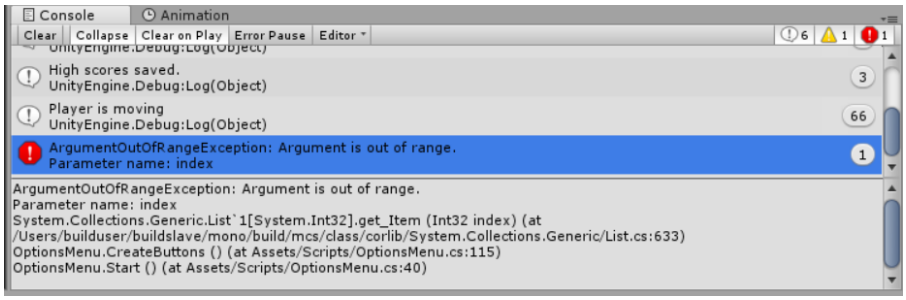
```
40    //This function generates a move by generating an operator and an integer
41    //that the operation will be performed on. The start number is modified by
42    //performing the opposite operator on it. When the operator chosen is
43    //multiplication, the operator number can only be a factor of the current
44    //target to keep all the sums using integer values.
45    int[] GenerateMove (int currentTarget) {
46        int operatorsNumber = Random.Range(1, currentTarget+1);
47        int operation = Random.Range(0, PersistentGameData.operatorCodesArray.Count);
48        char thisOperator = PersistentGameData.operatorCodesArray[operation];
49        switch (thisOperator) {
50            case '+':
51                startNumber = targetNumber - currentTarget;
52                break;
53            case '-':
54                startNumber = targetNumber + currentTarget;
55                break;
56            case '÷':
57                startNumber = targetNumber * currentTarget;
58                break;
59            default:
60                if ((operatorsNumber % currentTarget) != 0) {
61                    operatorsNumber = FindIntegerDivisor(currentTarget);
62                }
63                startNumber = targetNumber / currentTarget;
64                break;
65        }
66        int operatorUnicode = thisOperator;
67        int[] move = new int[] {operatorUnicode, operatorsNumber};
68        return move;
69    }
```

[Figure 235]

```
71    //Finds the prime factors of a number and multiplies a combination of them to find
72    //a integer number that goes into the target.
73    int FindIntegerDivisor (int target) {
74        List<int> primeList = new List<int>();
75        //Find the prime factors that are 2.
76        while ((target % 2) == 0) {
77            primeList.Add(2);
78            target /= 2;
79        }
80        //target is now an odd number
81        for (int i = 3; i <= Mathf.Sqrt(target); i += 2) {
82            //While i is a factor of target, add i to prime list and divide by target.
83            while ((target % i) == 0) {
84                primeList.Add(i);
85                target /= i;
86            }
87        }
88        //If target is a prime number greater than 2.
89        if (target > 2) {
90            primeList.Add(target);
91        }
92        //From the list of prime factors pick randomly between 1 and the number of factors
93        //and multiply them together.
94        int result = 1;
95        int numberOfFactorsMultiplied = Random.Range(1, primeList.Count);
96        for (int j = 1; j <= numberOfFactorsMultiplied; j++) {
97            int randomListPos = Random.Range(0, primeList.Count);
98            result *= primeList[randomListPos];
99        }
100       //Return the result
101       return result;
102   }
103 }
```

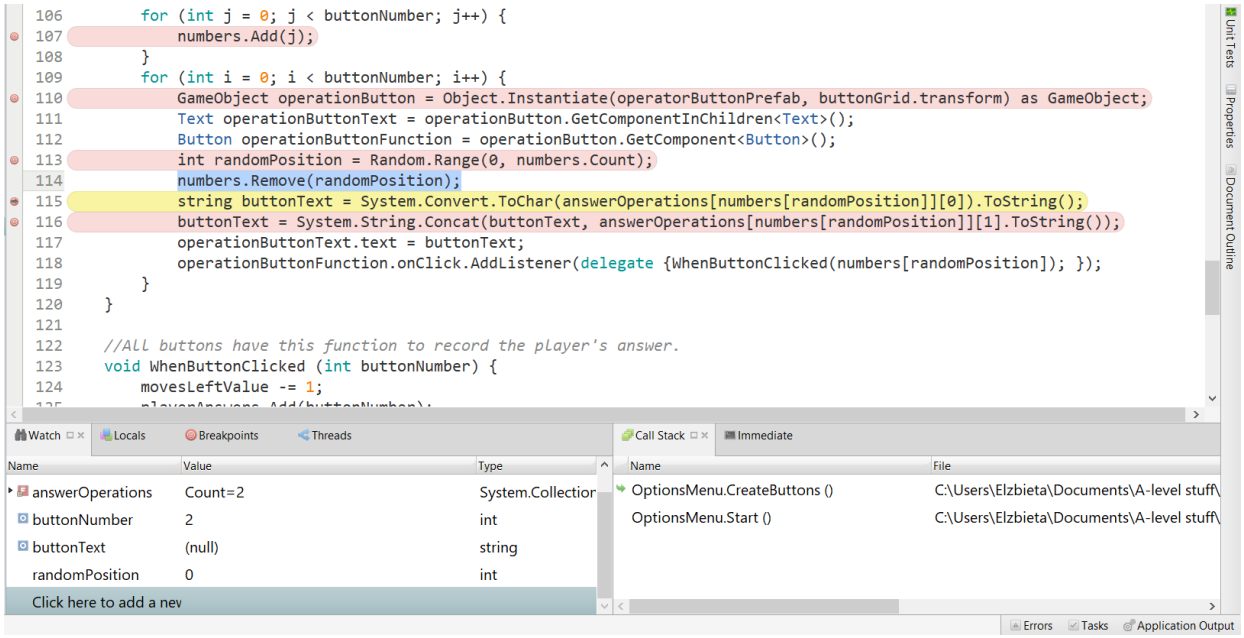
I then tested my game again. It still didn't work properly because of an error from the OptionsMenu class (**figure 236, page 124**) but there were no errors from the GenerateMathQuestion class.

[Figure 236]



I decided to use break points to try and find out what was causing the error in the OptionsMenu class. I found out it was because I was removing the randomPosition from the numbers before it was used when it should be removed after (**figure 237, page 124**).

[Figure 237]



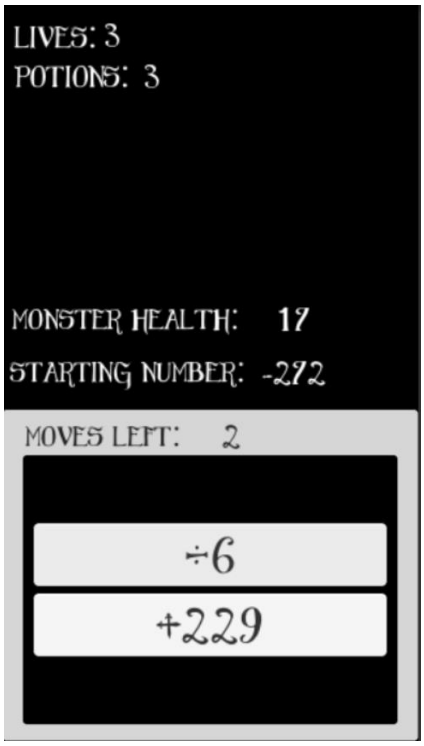
I moved the line of code to the end of the for loop but then I realised that it woudn't always remove the correct number so I changed the all the references to numbers[randomPositon] inside the for loop and changed numbers.Remove to numbers.RemoveAt (**figure 238, page 124**).

[Figure 238]

```
        for (int i = 0; i < buttonNumber; i++) {
            GameObject operationButton = Object.Instantiate(operatorButtonPrefab, buttonGrid.transform) as GameObject;
            Text operationButtonText = operationButton.GetComponentInChildren<Text>();
            Button operationButtonFunction = operationButton.GetComponent<Button>();
            int randomPosition = Random.Range(0, numbers.Count);
            int thisPosition = numbers[randomPosition];
            numbers.RemoveAt(randomPosition);
            string buttonText = System.Convert.ToChar(answerOperations[thisPosition][0]).ToString();
            buttonText = System.String.Concat(buttonText, answerOperations[thisPosition][1].ToString());
            operationButtonText.text = buttonText;
            operationButtonFunction.onClick.AddListener(delegate {WhenButtonClicked(thisPosition); });
        }
    }
```

I ran my game again and this time there were no errors from the console but when the question was generated the button was showing the wrong values to complete the calculation (**figure 239, page 125**).

[Figure 239]



It seemed to me like there was a button that was missing so I used breakpoints again to try debugging the problem. The problem was with the GenerateMathQuestion class because it wasn't generating proper questions because it was using the wrong variables to do calculations. I changed the GenerateMove method so that it would now properly generate questions (**figure 240, page 125**).

[Figure 240]

```
40    //This function generates a move by generating an operator and an integer
41    //that the operation will be performed on. The start number is modified by
42    //performing the opposite operator on it. When the operator chosen is
43    //multiplication, the operator number can only be a factor of the current
44    //target to keep all the sums using integer values.
45    int[] GenerateMove (int currentTarget) {
46        int operatorsNumber = Random.Range(1, currentTarget+1);
47        int operation = Random.Range(0, PersistentGameData.operatorCodesArray.Count);
48        char thisOperator = PersistentGameData.operatorCodesArray[operation];
49        switch (thisOperator) {
50            case '+':
51                startNumber -= operatorsNumber;
52                break;
53            case '-':
54                startNumber += operatorsNumber;
55                break;
56            case '÷':
57                startNumber *= operatorsNumber;
58                break;
59            default:
60                if ((operatorsNumber % currentTarget) != 0) {
61                    operatorsNumber = FindIntegerDivisor(currentTarget);
62                }
63                startNumber /= operatorsNumber;
64                break;
65        }
66        int operatorUnicode = thisOperator;
67        int[] move = new int[] {operatorUnicode, operatorsNumber};
68        return move;
69    }
```

I tested my game again and the error was finally fixed (**figure 241, page 126**).

[Figure 241]

Now there was just a problem with the potions because the player was receiving too many as you can see in **figure 241 (page 126)** where the player has 366 potions. This was because the number of potions the player receives gets increased in the Update method which is called once every frame and so the potions the player receives increases all the time while the player is correct which lasts for 2 seconds so the player receives around 120 potions during that time. I edited the code in the Update (**figure 242, page 126**) and DealWithOutcome (**figure 243, page 127**) methods from the OptionsMenu class that controls this to make sure it only gets called once.

[Figure 242]

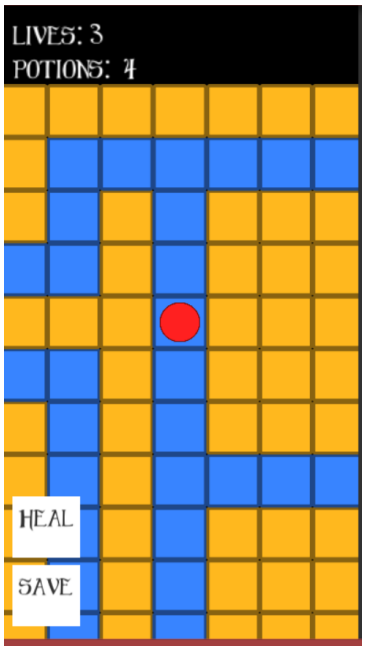
```
43    //Check if the player has answered the question or died.
44    void Update () {
45        if (movesLeftValue == 0) {
46            bool correct = true;
47            for (int i = 0; i < playerAnswers.Count; i++) {
48                if (playerAnswers[i] != i) {
49                    correct = false;
50                }
51            }
52            if (correct) {
53                //Player has won battle.
54                menu3.SetActive(true);
55                messageDisplay.text = "You have won! You received a potion as a reward.";
56                state = 1;
57                Invoke("DealWithOutcome", 2);
```

[Figure 243]

```
82    //The outcome of the player answering a question or dying.
83    void DealWithOutcome () {
84        switch (state) {
85            case 1:
86                if (giveRewards == true) {
87                    PersistentGameData.potions += 1;
88                    PersistentHighScores.enemiesDefeated += 1;
89                    giveRewards = false;
90                }
91                lvlManager.LoadAfterFade("04Dungeon");
92                break;
93            case 2:
94                menu3.SetActive(false);
95                menu2.SetActive(false);
96                menu1.SetActive(true);
97                break;
98            case 3:
99                SaveGameSystem.DeleteSaveGame("SavedGameData");
100               PersistentGameData.saveTheGame = false;
101               lvlManager.LoadAfterFade("01StartMenu");
102               break;
103       }
104   }
```

I tested this by creating a new game and now it worked (**figure 244, page 127**).

[Figure 244]



Scene 5 Review

This scene was also tricky to develop but I managed to complete it and now the success criteria 1.7, 1.8, 1.9, 1.10, 2.10, 2.11, 2.12, 2.13, 2.14, 2.15 and 2.16 have been met. While testing this scene, I tested my whole game because this was the only way to access the battle scene without loading it directly which would not generate proper questions because the questions are generated based on the settings the user chose at the start of the game. I changed the code I had written for previous scenes a lot especially what was saved by my save game system. Although I created the basic interface for this scene I have still not added the completed user interface which I will add during the final development stage before testing.

Final adjustments before testing the whole game

I removed the splash scene from my game since it no longer held any purpose and made the game take longer to load the Start Menu scene and I also removed it from the build settings.

I also realised that my game could crash because of the continue game button on the Start Menu because it will always load the Dungeon scene even if there is no saved game, so I created a new class to check there was a saved game and only load the game if there was a saved game (**figure 245, page 128**).

[Figure 245]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ContinueGame : MonoBehaviour {

    private LevelManager lvlManager;

    //Find the Level manager
    void Start () {
        lvlManager = GameObject.FindObjectOfType<LevelManager>();
    }

    //Called if the user selects continue game and only loads the game if there is a file saved.
    public void ContinueGameSelected () {
        if (SaveGameSystem.DoesSaveGameExist ("SavedGameData")) {
            lvlManager.LoadAfterFade("04Dungeon");
        }
    }
}
```

I then tested it to make sure it worked by creating a new game and dying to delete my save and then I tried to press the continue game button. The test was successful because when I pressed the continue game button, nothing happened or was loaded.

I also edited the GenerateMathQuestion class to make sure the least number of moves was at least 2 instead of 1 (**figure 246, page 128**) because when it was 1, all the player needed to do was select the operation shown to them without doing any mental maths.

[Figure 246]

```
    //This procedure performs all the required steps to generate a maths
    //question such as generating a target number and the number of moves
    //the question will take to solve. It generates each move and adds
    //them to a sequence to be passed to the options class.
    void GenerateQuestion () {
        answerSequence = new List<int[]>();
        targetNumber = Random.Range(PersistentGameData.rangeMin, PersistentGameData.rangeMax + 1);
        numberOfMoves = Random.Range(2, PersistentGameData.maxMoves +1);
        startNumber = targetNumber;
        for (int moveNumber = 0; moveNumber < numberOfMoves; moveNumber++) {
            int[] moveGenerated = GenerateMove(startNumber);
            answerSequence.Add(moveGenerated);
        }
    }
```

I changed the user interface on the battle and dungeon scenes so that they were now the final versions. I added a dungeon level label to the dungeon scene and replaced the test sprites on the tiles with texture sprites and I changed the character of the player, so they are now a knight.

I added the image of the monster the player will battle to the battle scene as well as the player image.

I also added a scene to show the rules of the game which my stakeholders asked for.

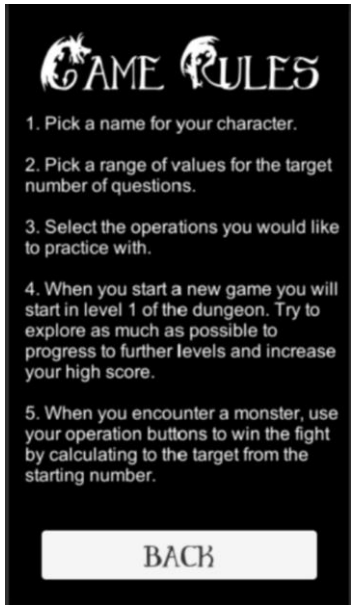
Name: Elzbieta Stasiak
Candidate Number: 5053

Centre Name: Coloma Convent Girl's School
Centre Number: 14310

My final scenes look as follows:

[The Start Menu scene]

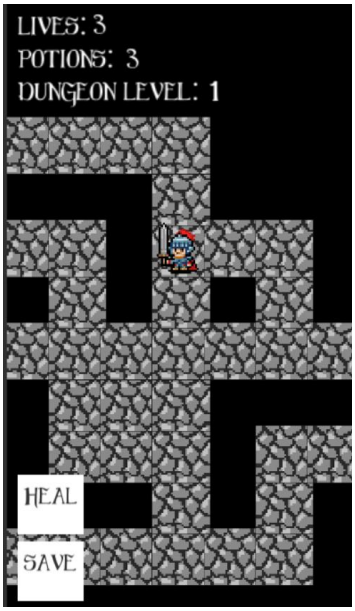
[The game rules scene]



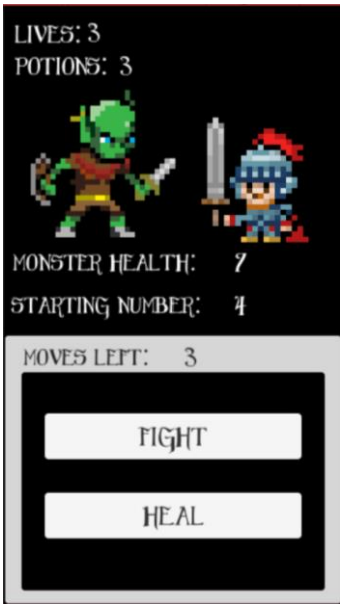
[The high scores scene]

[The New Game scene]

[The Dungeon scene]

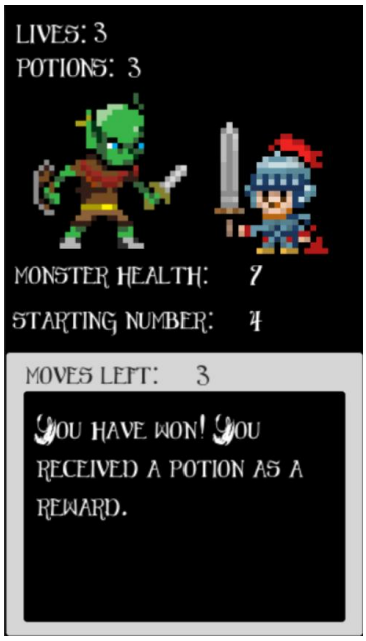


[Menu 1 of the battle scene]

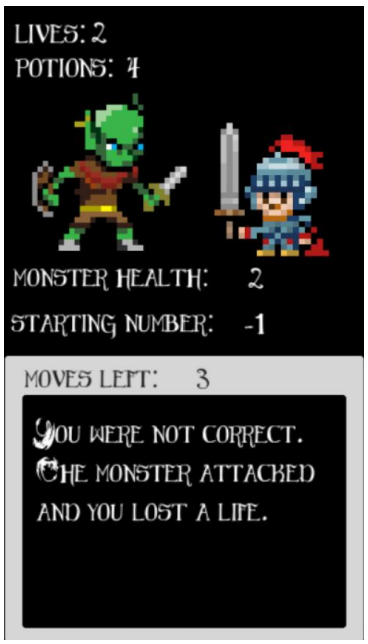


[Menu 2 of the battle scene]

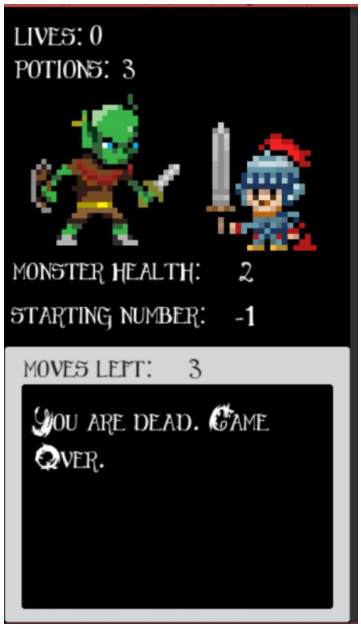
[The player winning a fight]



[The player losing a fight]



[The player dying]



I then went back to my interface and functionality success criteria to check each had been fulfilled. Because I made sure each scene functioned properly without any bugs before I continued development to the next one I was sure that the evidence provided during development was enough for each success criteria.

Interface success criteria evidence

	Criteria	Evidence	Location of first evidence
1.1	The Start Menu has a menu with the buttons: new game, continue game and view high scores.	Screenshot of the first screen when the game is loaded.	Figure 5, page 34

1.2	The first three scenes have headings differentiating them	Screenshot of the first 3 scenes with headings	Figures 5, 6 and 7, page 34
1.3	My game uses simple 2D graphics to display the dungeon and character.	Screenshot of the dungeon scene showing the player character in the dungeon.	Figure 183, page 102
1.4	All buttons and input fields have to be able to be interacted with using touch controls.	Screenshot of buttons being used from touch controls.	Figures 8, 9 and 10, page 35
1.5	The player is able to touch the tile they want the player to move to and their character will move to the corresponding location in the dungeon.	Screenshot of the player character moving to the position touched on screen.	Figure 195, page 106
1.6	The player will be able to view their character and the monster they are battling when they encounter a monster.	Screenshot of the player in a battle with a monster.	Menu 1 of the battle scene, page 129
1.7	The player will be able to view their character's health and number of potions while on the dungeon or battle screen.	Screenshots of the dungeon and battle screens with the player's health and potions shown.	Figure 157, page 93
1.8	The player will be able to view the starting number and target number of each maths question.	Screenshots of the battle screen with the player's starting and target number shown.	Menu 2 of the battle scene, page 130
1.9	The player will be able to view and select the operations they want to use to solve the maths question.	Screenshots of the battle screen with the operation buttons shown.	Menu 2 of the battle scene, page 130
1.10	The player will be able to view the amount of moves they have to solve a question which will decrease every time the user picks an operation button.	Screenshots of the battle screen with the amount of moves left shown.	Menu 2 of the battle scene, page 130

Functionality success criteria evidence

	Criteria	**Evidence**	**Page location of first evidence**
2.1	Start Menu is the first screen to load	Screenshot of the first screen when the game is loaded.	Figure 5, page 34
2.2	The continue game button loads the main dungeon screen with the player's last saved game.	Screenshot of the start menu with a continue game button.	The start menu scene, page 129
2.3	The high scores button takes the player to the high scores screen.	Screenshot of the start menu with a high scores button.	Figures 56, 57 and 58, page 47
2.4	The New game screen loads when the user chooses new game button from the start menu.	Screenshot of the new game screen with options and buttons to start the game and go back to the start menu.	The new game scene, page 129

2.5	When on the high scores scene, the high scores are loaded from a save file and displayed to the use.	Screenshot of the high scores screen with the saved high scores viewed.	Figures 56, 57 and 58, page 47
2.6	The high scores of the game need to be saved	Screenshot of the high scores screen with the saved high scores viewed.	Figures 56, 57 and 58, page 47
2.7	The dungeons are randomly generated.	Screenshots of randomly generated dungeons.	Figure 154, page 92
2.8	All the dungeon levels have an exit that takes the player to the next level of the dungeon.	Screenshot of a dungeon with as exit.	Figure 154, page 92
2.9	The dungeons get bigger and more complex as the level of the dungeon increases	Screenshot of the later dungeon levels.	Figure 130, page 83
2.10	Maths questions are generated based on the settings chosen by the user when they created a new game.	Screenshot of a maths question generated.	Menu 2 of the battle scene, page 130
2.11	With each question, the user will start with a starting number they are given, and they must use the buttons with operations like "+4" to reach a target number which will be the monsters health.	Screenshot of a maths question generated.	Menu 2 of the battle scene, page 130
2.12	The player will randomly encounter monsters in the dungeon while exploring.	Screenshot of the player encountering a monster.	Menu 1 of the battle scene, page 129
2.13	The player will be able to heal themselves by using a potion if they have one to give them full health.	Screenshot of the player healing themselves.	Menu 1 of the battle scene, page 129
2.14	If the player tries to incorrectly solve a maths question, the monster they are battling will attack them and the player will lose a life.	Screenshot of the player guessing the incorrect answer.	The player losing a fight, page 130
2.15	The player will die if they lose all their lives and they lose their progress and will have to start a new game.	Screenshot of the player after they have died.	The player dying, page 130
2.16	The player will gain a health potion from defeating a monster.	Screenshot of the player after they have defeated an enemy.	The player winning a fight, page 130
2.17	There will be a management system in my game.	Screenshots of the code of management system classes.	Figure 47, page 43

I have now met all my success criteria apart from my hardware success criteria which I will test by running my game on my phone.

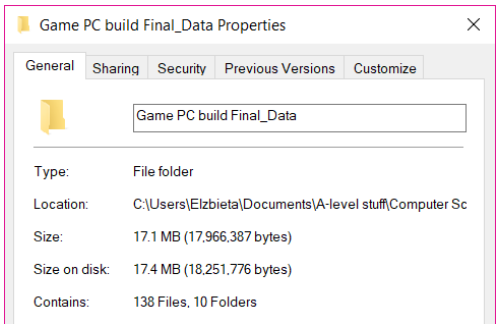
Testing the fully built version of my game

I was downloading the Java Development kit and the android SDK tools to test my game on my phone, but I couldn't get the android SDK tools file to run and install the packages needed to test my game, so I wasn't able to test the game on my phone.

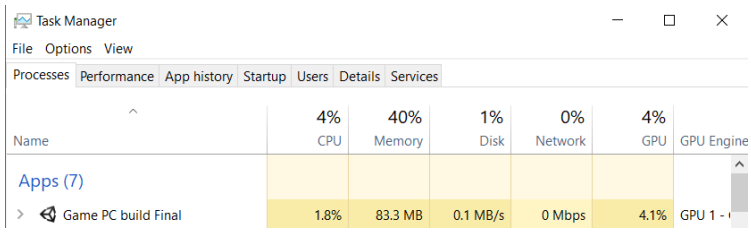
Instead I decided to build a version of my software that would be able to run on PC and I would use Windows task manager to measure the performance of my game although it wouldn't exactly be the same as running it on my phone.

When I initially built a version of my game the menus on the battle scene where the wrong size, so I changed their screen fit modes in Unity and built a new version. Because I originally designed my game for the size of a phone screen one difference that I couldn't change was the view of the tiles of the dungeon screen. The most I could do was run it at a lower screen resolution of 1024x768, but all the resolutions were for horizontal screens. Other than the different screen size it ran fine so I started testing the hardware of my game. My game took up a file size of about 18MB (**figure 247**) which is not very large for modern smartphones which have storage space of around 16 – 32 GB on average so I consider success criteria 3.2 met. When running the game, the game used around 90MB of RAM on average and never went above 100MB. It also only used up to 2.5% of the processing power of my PC which was mainly when anything was being generated such as the dungeon, a maths question or the player was moving which was after the pathfinding algorithm had been run (**figure 248**). My PC never needed to run at more than 1GHz when running the game (**figure 249**), so it does not take a lot of power to run and could probably run for a long time on a phone without using the battery. I will consider the success criteria 3.1 and 3.3 being met as when running the game on my PC, the game meets my criteria.

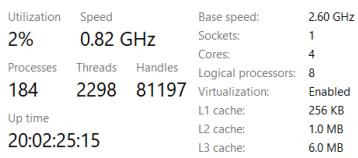
[Figure 247]



[Figure 248]



[Figure 249]



Beta testing with users

I tested the game on my laptop with my stakeholders. I managed to interview 4 people which was half of my class. I interviewed them while I watched them play the game and after they had finished testing I would ask them the questions:

1. How easy is the game to use and is it intuitive?
2. Would you prefer to use this game to improve your mental maths than another method?
3. Are there any features you would add to improve the game?
4. Are there any existing features you would improve?

Stakeholder Comments:

1. "The game is very easy to use and clear."
   "I think the font used is very cool."
2. "I would prefer to use your game since it is more fun than just doing questions and I like the way you can answer questions."
   "The game is fun especially if you like maths."
   "I liked randomly encountering monsters as it feels nostalgic and like games like Dragon Quest and Pokémon."
3. "You could add microtransactions to get more potions."
   "You could add chests that you could find in the dungeon to give you more potions."
4. "I liked the images used for the character and the monster, but you could add animation to them."
   "You could add the option to see your answer calculation being performed on the monster's health to see where you went wrong."

Overall, all interviewees agreed the controls were easy to understand although some didn't realise that they could select tiles that were more than one tile away from where their character was, in the beginning to move to the tile. When interviewees entered an incorrect input on the new game screen, the dialog box appeared and told them what was wrong, and they understood how to enter valid data from the message it displayed. All interviewees also agreed they would prefer to use my game to practice their mental maths. They liked the ability to move around in the dungeon and randomly encounter monsters as they found it very similar to other games. They liked the system used to answer questions and thought it was more interesting than just having one sum to do. They also liked the choice at the start of the game to choose what they wanted to practice. One feature an interviewee suggested was to include microtransactions to gain more potions in game, so the player could survive for longer or add chests that the player could find potions in. Most interviewees agreed that the visuals in the game could do with improvement as they were quite simple, but they liked the font used by most of the game. During testing there would sometimes be an issue where the maths question would not be generated properly which didn't allow it to be solved and the game would end as the player would lose all their lives.

After the beta testing I found what the issue was and fixed it. It was a problem with the Options menu class where the list containing the player's answers wasn't cleared after they answered a question wrong (**figure 250**).

[Figure 250]

```
82    //The outcome of the player answering a question or dying.
83    void DealWithOutcome () {
84        switch (state) {
85            case 1:
86                if (giveRewards == true) {
87                    PersistentGameData.potions += 1;
88                    PersistentHighScores.enemiesDefeated += 1;
89                    giveRewards = false;
90                }
91                lvlManager.LoadAfterFade("04Dungeon");
92                break;
93            case 2:
94                playerAnswers = new List<int>();
95                menu3.SetActive(false);
96                menu2.SetActive(false);
97                menu1.SetActive(true);
98                break;
99            case 3:
100               SaveGameSystem.DeleteSaveGame("SavedGameData");
101               PersistentGameData.saveTheGame = false;
102               lvlManager.LoadAfterFade("01StartMenu");
103               break;
104       }
105   }
```

Evaluation

Evaluating my game using my success criteria

Interface criteria

The evidence found for each individual success criteria is referenced in the interface table on **pages 131 to 132**.

- **1.1 The Start Menu has a menu with the buttons: new game, continue game and view high scores.**
  It is clear to the user what they can do when they open the app, I also added a game rules button to the start menu that my stakeholders wished for so that new players understand how the game works.
- **1.2 The first three scenes have headings differentiating them.**
  The scenes have clear headings and my stakeholders understood which scene was which.
- **1.3 My game uses simple 2D graphics to display the dungeon and character.**
  The player can see a clear map of the dungeon and their player and can tell where they can walk to explore the dungeon.
- **1.4 All buttons and input fields have to be able to be interacted with using touch controls.**
  This was only partly met as I could not test my game on my phone, but all the buttons can be interacted with using just one click which is equivalent to tapping them. I am assuming that when any input fields are interacted with, a phone's keyboard should be triggered to appear to let the user type their input as I made them using Unity's input field component which is cross-platform. I tested that the input was validated properly and told the user if any of their inputs were not acceptable using a dialog box.
- **1.5 The player is able to touch the tile they want the player to move to and their character will move to the corresponding location in the dungeon.**
  When the player touches a tile, their character immediately travels to it and there is no delay between touching a tile and the player walking towards it. The player takes the most efficient route and the camera follows them smoothly through the dungeon.
- **1.6 The player will be able to view their character and the monster they are battling when they encounter a monster.**

The player can see the monster they are battling. Currently I only added one monster character to the game but I would like to add more in the future so there is more variety in creatures as well as maybe add other designs for the player character.

- **1.7 The player will be able to view their character's health and number of potions while on the dungeon or battle screen.**
  The player can see the number of potions they have left when in the dungeon or in a battle. They can tell how close they are to dying from the number of lives they have left and if they need to use a potion to refill their health.
- **1.8 The player will be able to view the starting number and target number of each maths question.**
  The player is shown the starting number they are given when they get into a battle and the target number is shown as the health of the monster which the player needs to calculate in their head using the operations given to them to defeat the monster.
- **1.9 The player will be able to view and select the operations they want to use to solve the maths question.**
  The player is shown each operation as a separate button which they can press to input their answer. Each operation shows one of the operators the player has chosen at the start of their game and the value of the operation such as '+4'. The menu system that holds the buttons adjusts its size to fit the number of buttons generated so if there are more than can fit on the screen the menu becomes scrollable, so the player can access all the buttons.
- **1.10 The player will be able to view the amount of moves they have to solve a question which will decrease every time the user picks an operation button.**
  The player always knows how many moves they get to use and the number updates as the player picks operations to use.

In summary, all of the interface success criteria have been met apart from the interface being tested on a phone because I wasn't able to get the Android SDK running (**page 132**). I am quite sure it works however as games made with Unity are meant to be cross-platform, so the interface should work. The only part of my interface I didn't get to test was the pinch zoom function (**page 108**) as this only worked on touch screens unlike moving the player character where a touch could be replaced by a mouse click.

Functionality criteria

The evidence found for each individual success criteria is referenced in the functionality table on **pages 132 to 133**.

- **2.1 Start Menu is the first screen to load.**
  Since I made this menu the first in the build settings order, it will always load first allowing the player access to their high scores, the game rules, the settings to create a new game and their saved game.
- **2.2 The continue game button loads the main dungeon screen with the player's last saved game.**
  If the player has a game saved, the continue game will load the player's game. If the player has never created a game before, they cannot continue a game and if the player dies they cannot reload their save and try again.
- **2.3 The high scores button takes the player to the high scores screen.**
  The player can view their high scores by pressing the high scores button on the start menu.

- **2.4 The new game screen loads when the user chooses the new game button from the start menu.**
  The new game screen can be accessed by pressing the new game button on the start menu. On the new game screen, they can then decide the settings for their game.
- **2.5 When on the high scores scene, the high scores are loaded from a save file and displayed to the user.**
  The player can keep track of how well they are doing by viewing the high scores and can see what they need to beat. All the high scores are updated as the user plays the game if they beat them. The only one that isn't updated is the bosses defeated as I didn't end up adding any boss monsters to my game, but in future I would like to.
- **2.6 The high scores of the game need to be saved.**
  The high scores are loaded from a file that keeps the high scores of every game the player has ever played. This file is separate from the saved game data file that stores the user's settings for each game and can only store the settings of one game at a time. The file is saved as a binary serialized file, so the user cannot cheat by trying to change the data like they could with a text file.
- **2.7 The dungeons are randomly generated.**
  This part of my game was the hardest to create and required the most testing, but it was also the most interesting and I am happy with the layouts of the dungeons created. Because they are randomly generated, some of the maps are very easy to navigate and some are harder, but they are all traversable and fun to explore. In my initial testing, the larger dungeons took longer to create but when testing the game all were generated almost immediately, so I no longer had to worry about the game not responding for a while.
- **2.8 All the dungeon levels have an exit that takes the player to the next level of the dungeon.**
  Progressing to further dungeon levels increases the difficulty of the maths questions by increasing the maximum number of moves a question can take. Because the position of an exit and the spawn point of the player are random within the dungeon. Sometimes the player can spawn right next to the dungeon exit but most of the time this doesn't happen.
- **2.9 The dungeons get bigger and more complex as the level of the dungeon increases.**
  The increase in dungeon level makes the dungeons larger. This means they have more and bigger rooms so there are more corridors and places where the exit could be.
- **2.10 Maths questions are generated based on the settings chosen by the user when they created a new game.**
  The settings the user has chosen include the operations they get to use as well as the range of numbers for the target number which is the monster's health. Because I generate the questions starting from the target number, the starting number and values used for the operations can sometimes be above or below this range, but this could be improved by allowing the user to pick separate ranges for these values.
- **2.11 With each question, the user will start with a starting number they are given, and they must use the buttons with operations like "+4" to reach a target number which will be the monsters health.**
  Each question is different so there is an almost infinite number of questions the user could come across which is great for people who want to keep replaying the game and practice their mental maths.

- **2.12 The player will randomly encounter monsters in the dungeon while exploring.**
  The player doesn't know when to expect being asked a question as there is a 1 in 18 chance when they step on a tile, they will encounter a monster. This allows players some time to explore the dungeon before doing the next question, so the player doesn't get tired from having to answer questions repeatedly.
- **2.13 The player will be able to heal themselves by using a potion if they have one to give them full health.**
  The player gets more chances to answer questions and receives a potion every time they defeat a monster as a reward.
- **2.14 If the player tries to incorrectly solve a maths question, the monster they are battling will attack them and the player will lose a life.**
  The player has a penalty for answering questions wrongly and the game is not stressful from giving you a time limit. The player can use potions to refill their lives, but they only have a limited supply of them, so they don't get endless chances.
- **2.15 The player will die if they lose all their lives and they lose their progress and will have to start a new game.**
  The player has a penalty for answering too many questions wrong, so to improve their high scores they must keep practicing and get better at their mental maths. This also means the player can change the settings of their game if they find it too hard and end up dying too quickly.
- **2.16 The player will gain a health potion from defeating a monster.**
  This is the reward for defeating a monster. The quicker a player defeats a monster, the more likely they are to survive and increase their chances of survival.
- **2.17 There will be a management system in my game.**
  The management of my game is spread across different classes in different scenes with different purposes. Across all the scenes, the level manager controls the loading of different scenes and the save game system controls loading and saving data to permanent files. Each scene from the new game scene onwards, has it's own management class to manage input and coordinate other classes to work together on the other scenes such as the dungeon manager.

Hardware criteria

The evidence from testing the hardware is on **page 134**. The hardware success criteria were partially met for the same reason as the interface success criteria as I was unable to test my game on my phone, so I was only able to gather approximate data from the game running on my PC.

- **3.1 The game must be able to be run smoothly using the computational power of an average modern Android phone.**
  On my PC, my game ran without using a lot computational power and power in the range of modern smart phones. It ran very smoothly with no crashing or freezing in the final game.
- **3.2 The game must use as little memory as possible.**
  The game uses less than 100MB of RAM when running so it should run fine on modern phones without using up too much of the couple of GB a phone typically has of RAM.
- **3.3 The game must run using as little battery power as possible.**
  I couldn't really test this without having the game running for long periods of time on my phone but since it didn't use up a lot computational power to run, I think it should be able to be run for a suitable amount of time on a phone although this does depend on the particular phone's battery life.

Evaluating the usability of my game

On my PC, my stakeholders had no major problems with the controls or layout of my game (**page 134**). They liked the font I'd chosen to use for most of the game and had no trouble reading it. They liked that it was simple to use and needed almost no help interacting with the game. The game rules I added helped my stakeholders understand how the game worked. However, if I could have tested it using my phone, I would have liked to have tested whether my stakeholders found the text format easy to read and whether the buttons were obvious to press while using the smaller screen of a phone. As well as the tile size being big enough to select individual tiles and the pinch zoom feature having an acceptable zoom speed.

Issues for future development

If I could continue development of my game in the future I would make sure my game is bug free and the interface has more explanation of its different functions. I would also retry trying to get my game to run on my phone maybe using a different PC or waiting for a new version of Android SDK to be released and test it then.

The generation of the maths questions could be improved such as tweaking the maximum number of moves because rarely, questions with 7 moves would be generated with multiple buttons being identical which made it almost impossible to solve these questions. I would change my system so that only none identical operations were generated and possibly limit the maximum number of moves more. I would also remove any operations that were multiply or divide by 1 because they are pointless at testing your mental maths (**figure 251**). I would also change my system so that it limits what the starting number can be as currently there is no limit on it and it ended up being sometimes bigger than the target number limit the user had set or a negative number because the system I created worked backwards from the target number to generate the operation buttons and the start number.

[Figure 251]

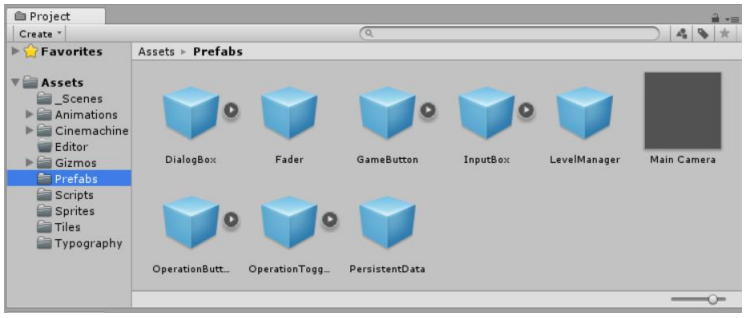
I would like to improve the graphics and overall quality of my game by adding possibly more tile types other than the simple floor and exit tiles and also allowing the player to customise their character. I would also add different monster types to make the game more varied. I would add background music to my game, so the user doesn't have to play the game in silence. I would also add extra sound effects to improve the experience.

Future maintenance of my game

I added comments throughout my code to aid future development which should hopefully be enough for other developers to understand my code. I could have added more but then I would find my code too hard to read. I also could have annotated the variables used so that it was obvious what they were for, but I think my labelling of variables should be enough in most situations. This may get confusing however, when many variables are related such as the many dungeon variables which had similar names.

So I didn't have to create multiple copies of game objects, the ones that I used repeatedly like the game objects that hold the persistent data classes or the level manager, I made prefabs of (**figure 252**). Unity has a very robust prefab system that helped me because it allows you to save copies of game objects which you can add to a scene and make each one slightly different like the operator buttons which are all instantiated from the same prefab but display different operation. Prefabs also enabled me to make changes to one game object and apply it to all others such as when changing the graphics of the tiles, I just changed the image attached to the tile prefab once and all tiles generated after used the same image.

[Figure 252]



In the future, my stakeholder's requirements may change for example if they wanted more operations to be added such as using indices they would not be too difficult to implement as my program is very modular with each class having its own function so the only parts that would have to change would be adding the option to the list of operations the user has to select on the Create New Game screen and editing the CreateNewGame and GenerateMathQuestion so that they can accept this new operation and generate questions with it. All other parts of my game would stay the same.

Improving the graphics would not affect the game's code as the graphics and code are separate. My code only deals with the overall game object, but all game objects are made up of multiple components, with the image renderers being one component that can easily have its data changed such as changing the sprites, which I actually did when making final adjustments and I didn't have to edit any of my code to do it.

In conclusion, this project was very large and took a lot of time test as there was a lot of random generation which meant it was very hard to test for every possibility but I learned a lot of new algorithms and coding techniques during the project and got to implement some of the algorithms I have learnt about during my Computer Science and Maths A-Levels such as the A* and Prim's Algorithm which improved my understanding of them as well as using debugging techniques such as break points.

Appendix

My blank mental maths google form

CS Coursework Stakeholder Research

Form description

How would you rate your mental maths skills from 1 - 10? *

	1	2	3	4	5	6	7	8	9	10	
No skill	○	○	○	○	○	○	○	○	○	○	Very skilled

How important do you find mental maths in your daily life from 1 - 10? *

	1	2	3	4	5	6	7	8	9	10	
Not important	○	○	○	○	○	○	○	○	○	○	Very important

How important do you find mental maths for maths exams from 1 - 10? *

	1	2	3	4	5	6	7	8	9	10	
Not important	○	○	○	○	○	○	○	○	○	○	Very important

Would you like to improve your mental maths? *

- ○ Yes
- ○ No

Have you tried using any phone apps to improve your mental maths? *

- ○ Yes
- ○ No

Have you tried using any phone apps to improve your mental maths? *

- Yes
- No

If you have, can you list any features you liked about the ones you tried?

Long answer text

Were there any features you didn't like?

Long answer text

The C# code of all the classes I wrote

This is the code of all my classes in alphabetical order of class name

ContinueGame

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ContinueGame : MonoBehaviour {

    private LevelManager lvlManager;

    //Find the level manager
    void Start () {
        lvlManager = GameObject.FindObjectOfType<LevelManager>();
    }

    //Called if the user selects continue game and only loads the game if there is a file saved.
    public void ContinueGameSelected () {
        if (SaveGameSystem.DoesSaveGameExist ("SavedGameData")) {
            lvlManager.LoadAfterFade("04Dungeon");
        }
    }
}
```

CreateNewGame

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class CreateNewGame : MonoBehaviour {

    public InputField playerName, maxRange, minRange;
    public GameObject dialogBox, errorMessage;

    private bool validSelection;
    private Toggle[] myToggles;
    private DialogErrorDisplay errorDisplay;
    private LevelManager lvlManager;
    //TODO private GenerateDungeon genDungeon
```

```
    //Finds all the classes it needs data from, including the PersistentGameData,
    //all the toggles, the level manager and the dungeon generator.
    void Start () {
        myToggles = GameObject.FindObjectsOfType<Toggle>();
        lvlManager = GameObject.FindObjectOfType<LevelManager>();
    }

    //Called when the player presses the Start Game button and reads what the
    //player has entered as well as creating new data to be saved.
    public void CreateGame () {
        bool correctInput = ValidateInputs();
        GetOperatorCodes();
        if (validSelection == false) {
            OpenDialogBox("Please pick at least one operator to use.");
        } else if (correctInput == true) {
            PersistentGameData.maxMoves = CalculateMaxMoves (PersistentGameData.dungeonLevel);
            PersistentGameData.dungeonMapSave = new int[1,1];
            PersistentGameData.playerPosSave = new Vector3(0,0,0);
            PersistentGameData.exitPosSave = new Vector3(0,0,0);
            PersistentGameData.dungeonLevel = 1;
            PersistentGameData.playerLivesSave = 3;
            PersistentGameData.potions = 3;
            lvlManager.LoadNextAfterFade();
        }
    }

    //Checks all the user inputs to make sure they are valid.
    bool ValidateInputs () {
        //If the player hasn't entered a name.
        if (playerName.text.Length == 0) {
            OpenDialogBox ("You forgot to enter a name.");
        } else {
            PersistentGameData.playerName = playerName.text;
            //Parsing the text to convert it into integers.
            int workingMin;
            int workingMax;
            //Convert the text into integers.
            bool minResult = int.TryParse (minRange.text, out workingMin);
            bool maxResult = int.TryParse (maxRange.text, out workingMax);
            //If the player hasn't entered a range.
            if (minResult == false) {
                OpenDialogBox ("You forgot to enter a minimum value.");
            } else if (maxResult == false) {
                OpenDialogBox ("You forgot to enter a maximum value.");
            } else {
                //If the minimum value is greater than the maximum values.
                if (workingMax < workingMin) {
                    OpenDialogBox ("Your minimum value is greater than your maximum value, please swap
the values.");
                } else {
                    //If the minimum value is 0 or negative.
                    if (workingMin < 1) {
                        OpenDialogBox("Your minimum value must be greater than zero.");
                    } else {
                        //If all conditions are met, the values are saved.
                        PersistentGameData.rangeMin = workingMin;
                        PersistentGameData.rangeMax = workingMax;
                        return true;
                    }
                }
            }
        }
        return false;
    }

    //Displays the dialog box with the error message given to it.
    void OpenDialogBox (string message) {
        dialogBox.SetActive(true);
        errorDisplay = errorMessage.GetComponent<DialogErrorDisplay>();
        errorDisplay.DisplayError(message);
    }

    //Called when the user presses the CLOSE button on the dialog box to stop displaying the box.
```

```
    public void CloseDialogBox () {
        dialogBox.SetActive(false);
    }

    //A formula I came up with myself to increase the maximum moves for a maths problem as the player
progresses.
    int CalculateMaxMoves (int currentLevel) {
        float cLevel = (float)currentLevel;
        float fMoves = (4f * Mathf.Sqrt (cLevel)) / 1.5f;
        int moves = Mathf.RoundToInt(fMoves);
        return moves;
    }

    //The operatorCodesArray stores the different operations the player wants to use, 1 is addition, 2
 is
    //subtraction, 3 is multiplication and 4 is division.The different toggles corespond to different
    //operators.
    void GetOperatorCodes () {
        PersistentGameData.operatorCodesArray = new List<char>();
        validSelection = false;
        for (int i = 0; i <= 3; i++) {
            validSelection = true;
            if (myToggles[i].isOn == true) {
                switch (i) {
                    case 1:
                        PersistentGameData.operatorCodesArray.Add('+');
                        break;
                    case 2:
                        PersistentGameData.operatorCodesArray.Add('-');
                        break;
                    case 3:
                        PersistentGameData.operatorCodesArray.Add('×');
                        break;
                    default:
                        PersistentGameData.operatorCodesArray.Add('÷');
                        break;
                }
            }
        }
    }
}
```

DialogErrorDisplay

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class DialogErrorDisplay : MonoBehaviour {

    private Text errorDisplay;

    //Called by the CreateNewGame class to initialise the text component of the game object and pass
    //a message to the dialog box to display.
    public void DisplayError (string errorMessage) {
        errorDisplay = GetComponent<Text>();
        errorDisplay.text = errorMessage;
    }
}
```

DisplayDungeon

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Tilemaps;

public class DisplayDungeon : MonoBehaviour {

    public Tile wall, floor, exit;
    public GameObject ExitMapObject, player;
```

```
    public Vector3Int startingPosition;

    private Tilemap tileMap, exitMap;

    //Finds the Tilemap components of the game object this class is attached to and the other tile map
 game objects.
    void Start () {
        tileMap = GetComponent<Tilemap>();
        exitMap = ExitMapObject.GetComponent<Tilemap>();
    }

    //Loops through all tiles and sets the correct tile at the correct position on the tile map.
    public void DisplayTileGrid (bool isNewDungeon) {
        float totalElements = (float)PersistentGameData.dungeonMapSave.Length;
        int gridLength = Mathf.RoundToInt(Mathf.Sqrt(totalElements)) - 1;
        int[,] dungeonGrid = PersistentGameData.dungeonMapSave;
        int floorArea = gridLength * gridLength;
        Vector3Int[] floorTileCoords = new Vector3Int[floorArea];
        int newTile = 0;
        Vector3Int currentCoords = new Vector3Int(0,0,0);
        for (int y = 0; y <= gridLength; y++) {
            for (int x = 0; x <= gridLength; x++) {
                currentCoords.x = x;
                currentCoords.y = y;
                if (dungeonGrid[x,y] == 0) {
                    tileMap.SetTile(currentCoords, wall);
                } else {
                    tileMap.SetTile(currentCoords, floor);
                    floorTileCoords[newTile] = currentCoords;
                    newTile++;
                }
            }
        }
        //Picks the position of the dungeon exit and the player's starting position if a new dungeon i
s generated,
        //otherwise the exit is placed at the coordinates loaded from memory.
        if (isNewDungeon == true) {
            int exitPosition;
            exitPosition = Random.Range(0, newTile);
            exitMap.SetTile(floorTileCoords[exitPosition], exit);
            PersistentGameData.exitPosSave = (Vector3)floorTileCoords[exitPosition];
            int entrancePosition;
            do {
                entrancePosition = Random.Range(0, newTile);
                startingPosition = floorTileCoords[entrancePosition];
                player.transform.position = startingPosition;
                PersistentGameData.playerPosSave = startingPosition;
            } while (exitPosition == entrancePosition);
            PlayerMovement movement = player.GetComponent<PlayerMovement>();
            movement.GetNewDungeon();
        } else {
            Vector3Int exitCoords;
            exitCoords = Vector3Int.RoundToInt(PersistentGameData.exitPosSave);
            exitMap.SetTile(exitCoords, exit);
        }
    }

}
```

DungeonGenerator

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DungeonGenerator : MonoBehaviour {

    public int testLevelValue = 1;

    private int[,] tileArray;
    private int gridLength, newRegionNumber;
    private List<Vector2Int> frontierTilesList;
    private List<int[]> connectors;
```

```
    //This method is called whenever a new dungeon needs to be created.
    public void CreateNewDungeon () {
        newRegionNumber = 1;
        CreateGrid(PersistentGameData.dungeonLevel);
        PlaceRooms(PersistentGameData.dungeonLevel);
        frontierTilesList = new List<Vector2Int>();
        GenerateCorridors();
        connectors = new List<int[]>();
        ConnectAllRegions();
        RemoveDeadEnds();
        PrintDungeon();
        PersistentGameData.dungeonMapSave = tileArray;
    }

    //This method is just used for debugging to check the dungeon looks right.
    void PrintDungeon () {
        string printedGrid = "Grid:";
        for (int y = gridLength; y >= 0; y--) {
            string printedGridLine = "| ";
            for (int x = 0; x <= gridLength; x++) {
                printedGridLine = System.String.Concat(printedGridLine, tileArray[x,y].ToString(), " |
 ");
                if (x == gridLength) {
                    printedGrid = System.String.Concat(printedGrid, "\n", printedGridLine);
                }
            }
        }
        Debug.Log(printedGrid);
    }

    //Creates a square grid of wall tiles.
    void CreateGrid (int sideLength) {
        //The size of the grid increases as the player gets further, it must also always be an odd num
ber.
        sideLength = (2 * sideLength) + 13;
        tileArray = new int[sideLength, sideLength];
        gridLength = sideLength - 1;
        for (int y = 0; y <= gridLength; y++) {
            for (int x = 0; x <= gridLength; x++) {
                tileArray[x,y] = 0;
            }
        }
    }

    //Attemps to place a number of rooms randomly on the grid, the greater the level of the dungeon th
e more
    //rooms it will try to place
    void PlaceRooms (int roomAttempts) {
        roomAttempts = roomAttempts + 2;
        int oddCoords = (gridLength / 2) - 2;
        for (int i = 1; i <= roomAttempts; i++) {
            //Find a random coordinate to place the room at.
            int xPos = GetRandomOddNumber(oddCoords);
            int yPos = GetRandomOddNumber(oddCoords);
            //Generate a random size for the room.
            int maxRoomSize = Mathf.RoundToInt(oddCoords/2);
            int xLength = GetRandomOddNumber(maxRoomSize);
            int yLength = GetRandomOddNumber(maxRoomSize);
            if (xLength == 1) {
                xLength = 3;
            }
            if (yLength == 1) {
                yLength = 3;
            }
            //Check the room won overlap with any other rooms and place the room on the grid.
            bool placeRoom = true;
            for (int y = yPos; y < (yPos + yLength); y++) {
                if (y > (gridLength-1)) {
                    placeRoom = false;
                    break;
                }
                for (int x = xPos; x < (xPos + xLength); x++) {
                    if (x > (gridLength-1) || tileArray[x,y] != 0) {
```

```
                        placeRoom = false;
                        break;
                    }
                }
            }
            //Place the room if it didn't overlap.
            if (placeRoom == true) {
                for (int y = yPos; y < (yPos + yLength); y++) {
                    for (int x = xPos; x < (xPos + xLength); x++) {
                        tileArray[x,y] = newRegionNumber;
                    }
                }
                //Increase the number of the region so the next region has a different number.
                newRegionNumber++;
            }
        }
    }

    //Generates a random odd number between 1 and a maximum value.
    int GetRandomOddNumber (int maxRange) {
        int ranOddNum = Random.Range(0, maxRange+1);
        ranOddNum = (2*ranOddNum) + 1;
        return ranOddNum;
    }

    //Generates corridors that fill the spaces between the rooms using a maze generator based on Prim'
s algorithm.
    void GenerateCorridors () {
        //Find a closed tile to start the maze generator at.
        Vector2Int startPoint = AreAllOddTilesOpen();
        //Check all the odd tiles are open to finish running the maze generator.
        do { //Maze generator.
            //Open the tile with the starting tile coordinates.
            tileArray[startPoint.x,startPoint.y] = newRegionNumber;
            //Find frontier tiles.
            FindFrontierTiles(startPoint.x, startPoint.y);
            //While the list of frontier cells is not empty
            while (frontierTilesList.Count != 0) {
                //Pick a random frontier tile from the list of frontier tiles.
                int randomFrontier = Random.Range(0, frontierTilesList.Count);
                Vector2Int frontierTile = frontierTilesList[randomFrontier];
                //Find the neighbors of the fontier cell
                Vector2Int[] neighbours = GetCardinalTileCoords(frontierTile.x, frontierTile.y);
                List<Vector2Int> connectNeighbours = new List<Vector2Int>();
                for (int i = 0; i <= 3; i++) {
                    if (neighbours[i] != Vector2Int.zero && tileArray[neighbours[i].x, neighbours[i].y
] == newRegionNumber) {
                        connectNeighbours.Add(neighbours[i]);
                    }
                }
                //Pick a random neighbour and connect it to the frontier tile.
                int randomNeighbour = Random.Range(0, connectNeighbours.Count);
                Vector2Int pickedNeighbour = connectNeighbours[randomNeighbour];
                tileArray[frontierTile.x, frontierTile.y] = newRegionNumber;
                tileArray[pickedNeighbour.x, pickedNeighbour.y] = newRegionNumber;
                int connectingX = (frontierTile.x + pickedNeighbour.x) / 2;
                int connectingY = (frontierTile.y + pickedNeighbour.y) / 2;
                tileArray[connectingX, connectingY] = newRegionNumber;
                //Remove the frontier tile from the list of frontier tile.
                frontierTilesList.RemoveAt(randomFrontier);
                //Find the frontier tiles of the current frontier tile.
                FindFrontierTiles(frontierTile.x, frontierTile.y);
            }
            newRegionNumber++;
            startPoint = AreAllOddTilesOpen();
        } while (startPoint != Vector2Int.zero);
    }

    //Loops through all tiles with odd coordinates and returns the coordinate of the first tile that i
s still closed,
    //Otherwise it returns the coordinates of the origin.
    Vector2Int AreAllOddTilesOpen () {
        for (int y = 1; y < gridLength; y += 2) {
            for (int x = 1; x < gridLength; x += 2) {
```

```
                if (tileArray[x,y] == 0) {
                    Vector2Int emptyTile = new Vector2Int(x, y);
                    return emptyTile;
                }
            }
        }
        return Vector2Int.zero;
    }

    //Adds the frontier tiles around a tile with the coordinates given to the frontier tiles list if i
t is not
    //already in the list.
    void FindFrontierTiles (int xCoord, int yCoord) {
        Vector2Int[] NESWtiles = GetCardinalTileCoords(xCoord, yCoord);
        for (int i = 0; i <= 3; i++) {
            if (NESWtiles[i] != Vector2Int.zero && tileArray[NESWtiles[i].x, NESWtiles[i].y] == 0) {
                if (frontierTilesList.Contains(NESWtiles[i]) == false) {
                    frontierTilesList.Add(NESWtiles[i]);
                }
            }
        }
    }

    //Get the coordinates of each tile two away from the origional coordinate in every cardinal direct
ion.
    Vector2Int[] GetCardinalTileCoords (int xCoord, int yCoord) {
        Vector2Int[] NESWArray = new Vector2Int[4];
        NESWArray [0] = new Vector2Int (xCoord, yCoord + 2);
        NESWArray [1] = new Vector2Int (xCoord + 2, yCoord);
        NESWArray [2] = new Vector2Int (xCoord, yCoord - 2);
        NESWArray [3] = new Vector2Int (xCoord - 2, yCoord);
        //Get rid of coordinates that are outside the grid.
        for (int i = 0; i <= 3; i++) {
            if (NESWArray [i].x > gridLength || NESWArray [i].x < 0) {
                NESWArray [i] = Vector2Int.zero;
            } else if (NESWArray [i].y > gridLength || NESWArray [i].y < 0) {
                NESWArray[i] = Vector2Int.zero;
            }
        }
        return NESWArray;
    }

    //Connect all rooms and corridors together to form one region.
    void ConnectAllRegions () {
        FindAllConnectors();
        //Connect the regions until the list of connecting tiles is empty.
        while (connectors.Count != 0) {
            //Get a random connector tile and connect the two regions it connects.
            int[] randomConnector = connectors[Random.Range(0, connectors.Count)];
            int startingRegion = randomConnector[2];
            int connectingRegion = randomConnector[3];
            //Open the connector.
            tileArray[randomConnector[0],randomConnector[1]] = startingRegion;
            //Convert the region number of the newly connected region to the number of the starting re
gion.
            for (int y = 1; y < gridLength; y++) {
                for (int x = 1; x < gridLength; x++) {
                    if (tileArray[x,y] == connectingRegion) {
                        tileArray[x,y] = startingRegion;
                    }
                }
            }
            //Remove all extraneous connectors between the two regions that are now connected but give
 them
            //some chance of opening up.
            for (int j = 0; j < connectors.Count; j++) {
                if (connectors[j][2] == startingRegion && connectors[j][3] == connectingRegion) {
                    OpenUpChance(connectors[j][0], connectors[j][1], startingRegion);
                    connectors.Remove(connectors[j]);
                } else if (connectors[j][2] == connectingRegion && connectors[j][3] == startingRegion)
 {
                    OpenUpChance(connectors[j][0], connectors[j][1], startingRegion);
                    connectors.Remove(connectors[j]);
                }
```

```
            }
        }
    }

    //Finds tiles that could connect two different regions.
    void FindAllConnectors () {
        //Iterate through all tiles.
        for (int y = 1; y < gridLength; y++) {
            for (int x = 1; x < gridLength; x++) {
                //Only check blocked tiles.
                if (tileArray[x,y] == 0) {
                    //Check the tiles north and south for different regions.
                    if (tileArray[x,y+1] != 0 && tileArray[x,y-
1] != 0 && tileArray[x,y+1] != tileArray[x,y-1]) {
                        connectors.Add(new int[4] {x, y, tileArray[x,y+1], tileArray[x,y-1]});
                    //Check the tiles east and west for different regions.
                    } else if (tileArray[x+1,y] != 0 && tileArray[x-
1,y] != 0 && tileArray[x+1,y] != tileArray[x-1,y]) {
                        connectors.Add(new int[4] {x, y, tileArray[x+1,y], tileArray[x-1,y]});
                    }
                }
            }
        }
    }

    //Give a tile a 1 in 10 chance of opening up.
    void OpenUpChance (int xCoords, int yCoords, int connectToRegion) {
        int chance = Random.Range(1, 11);
        if (chance == 1) {
            tileArray[xCoords, yCoords] = connectToRegion;
        }
    }

    //The method that is called from the main CreateNewDungeon method and starts off the recursive
    //removal of dead ends.
    void RemoveDeadEnds () {
        Vector2Int deadEnd = FindDeadEnds();
        tileArray[deadEnd.x, deadEnd.y] = 0;
        deadEnd = CountWalls(deadEnd.x, deadEnd.y);
        RecursiveRemove(deadEnd);
    }

    //Recursivley call this method until there are no more deadEnds.
    void RecursiveRemove (Vector2Int currentTile) {
        Vector2Int nextDeadEnd = CountWalls(currentTile.x, currentTile.y);
        if (nextDeadEnd != Vector2Int.zero) {
            //Close the dead end.
            tileArray[currentTile.x, currentTile.y] = 0;
            RecursiveRemove(nextDeadEnd);
        } else {
            Vector2Int newDeadEnd = FindDeadEnds();
            if (newDeadEnd != Vector2Int.zero) {
                RecursiveRemove(newDeadEnd);
            }
        }
    }

    //Find a new dead end and return its coordinates, otherwise return (0,0).
    Vector2Int FindDeadEnds () {
        for (int y = gridLength-1; y >= 1; y--) {
            for (int x = 1; x < gridLength; x++) {
                if (tileArray[x,y] != 0) {
                    if (CountWalls(x,y) != Vector2Int.zero) {
                        Vector2Int newDeadEnd = new Vector2Int(x,y);
                        return newDeadEnd;
                    }
                }
            }
        }
        return Vector2Int.zero;
    }

    //If the given tile is a dead end, return the coordinates of the tile connecting it to the
    //rest of the dungeon, otherwise return the coordinates (0,0).
```

```
    Vector2Int CountWalls (int xCoord, int yCoord) {
        int wallCount = 0;
        Vector2Int openSide = new Vector2Int(0,0);
        Vector2Int[] NESWArray = new Vector2Int[4];
        NESWArray [0] = new Vector2Int (xCoord, yCoord + 1);
        NESWArray [1] = new Vector2Int (xCoord + 1, yCoord);
        NESWArray [2] = new Vector2Int (xCoord, yCoord - 1);
        NESWArray [3] = new Vector2Int (xCoord - 1, yCoord);
        foreach (Vector2Int v in NESWArray) {
            if (tileArray[v.x,v.y] == 0) {
                wallCount++;
            } else {
                openSide = v;
            }
        }
        if (wallCount == 3) {
            return openSide;
        } else {
            return Vector2Int.zero;
        }
    }
}
```

DungeonManager

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class DungeonManager : MonoBehaviour {

    public GameObject livesDisplay, potionsDisplay, dungeonLevelDisplay;

    private DungeonGenerator dungeonGen;
    private DisplayDungeon displayDungeon;
    private Text lives, potions, dungeonLevel;

    //Finds the other classes on the level and manages when a new dungeon is generated.
    void Start () {
        dungeonGen = GameObject.FindObjectOfType<DungeonGenerator> ();
        displayDungeon = GameObject.FindObjectOfType<DisplayDungeon> ();
        if (PersistentGameData.dungeonMapSave.Length == 1) {
            dungeonGen.CreateNewDungeon ();
            displayDungeon.DisplayTileGrid (true);
        } else {
            displayDungeon.DisplayTileGrid (false);
        }
        //Displays the number of lives and potions the palyer has.
        lives = livesDisplay.GetComponent<Text>();
        potions = potionsDisplay.GetComponent<Text>();
        dungeonLevel = dungeonLevelDisplay.GetComponent<Text>();
        dungeonLevel.text = PersistentGameData.dungeonLevel.ToString();
        lives.text = PersistentGameData.playerLivesSave.ToString();
        potions.text = PersistentGameData.potions.ToString();
    }

    //When the player presses the heal button, one potion is used to give the player full health.
    public void Heal () {
        if (PersistentGameData.potions > 0) {
            PersistentGameData.potions -= 1;
            PersistentGameData.playerLivesSave = 3;
            lives.text = PersistentGameData.playerLivesSave.ToString();
            potions.text = PersistentGameData.potions.ToString();
        }
    }
}
```

GenerateMathQuestion

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GenerateMathQuestion : MonoBehaviour {

    private int numberOfMoves, startNumber, targetNumber;
    private List<int[]> answerSequence;

    //This function is called by the options menu class to get the number of
    //moves, the target number and the starting number of the question this
    //class generates.
    public int[] GetMathsQuestion () {
        GenerateQuestion();
        int[] results = new int[3] {numberOfMoves, startNumber, targetNumber};
        return results;
    }

    //This function is called by the options class menu to get the answer
    //sequence of the question generated.
    public List<int[]> GetAnswerSequence () {
        return answerSequence;
    }

    //This procedure performs all the required steps to generate a maths
    //question such as generating a target number and the number of moves
    //the question will take to solve. It generates each move and adds
    //them to a sequence to be passed to the options class.
    void GenerateQuestion () {
        answerSequence = new List<int[]>();
        targetNumber = Random.Range(PersistentGameData.rangeMin, PersistentGameData.rangeMax + 1);
        numberOfMoves = Random.Range(2, PersistentGameData.maxMoves +1);
        startNumber = targetNumber;
        for (int moveNumber = 0; moveNumber < numberOfMoves; moveNumber++) {
            int[] moveGenerated = GenerateMove(startNumber);
            answerSequence.Add(moveGenerated);
        }
    }

    //This function generates a move by generating an operator and an integer
    //that the operation will be performed on. The start number is modified by
    //performing the opposite operator on it. When the operator chosen is
    //multiplication, the operator number can only be a factor of the current
    //target to keep all the sums using integer values.
    int[] GenerateMove (int currentTarget) {
        int operatorsNumber = Random.Range(1, currentTarget+1);
        int operation = Random.Range(0, PersistentGameData.operatorCodesArray.Count);
        char thisOperator = PersistentGameData.operatorCodesArray[operation];
        switch (thisOperator) {
            case '+':
                startNumber -= operatorsNumber;
                break;
            case '-':
                startNumber += operatorsNumber;
                break;
            case '÷':
                startNumber *= operatorsNumber;
                break;
            default:
                if ((operatorsNumber % currentTarget) != 0) {
                    operatorsNumber = FindIntegerDivisor(currentTarget);
                }
                startNumber /= operatorsNumber;
                break;
        }
        int operatorUnicode = thisOperator;
        int[] move = new int[] {operatorUnicode, operatorsNumber};
        return move;
    }

    //Finds the prime factors of a number and multiplies a combination of them to find
    //a integer number that goes into the target.
```

```
    int FindIntegerDivisor (int target) {
        List<int> primeList = new List<int>();
        //Find the prime factors that are 2.
        while ((target % 2) == 0) {
            primeList.Add(2);
            target /= 2;
        }
        //target is now an odd number
        for (int i = 3; i <= Mathf.Sqrt(target); i += 2) {
            //while i is a factor of target, add i to prime list and divide by target.
            while ((target % i) == 0) {
                primeList.Add(i);
                target /= i;
            }
        }
        //If target is a prime number greater than 2.
        if (target > 2) {
            primeList.Add(target);
        }
        //From the list of prime factors pick randomly between 1 and the number of factors
        //and multiply them together.
        int result = 1;
        int numberOfFactorsMultiplied = Random.Range(1, primeList.Count);
        for (int j = 1; j <= numberOfFactorsMultiplied; j++) {
            int randomListPos = Random.Range(0, primeList.Count);
            result *= primeList[randomListPos];
        }
        //Return the result
        return result;
    }
}
```

HighScores

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class HighScores : MonoBehaviour {

    private GameObject[] textBoxes;
    private Text[] myText;

    //Finds the text boxes that display the high scores and finds the text components of them as well
as the saved high
    //scores.
    void Start () {
        //Only the text boxes that are being used to display the high scores have been given the 'Disp
layBox' tag.
        textBoxes = GameObject.FindGameObjectsWithTag ("DisplayBox");
        myText = new Text[textBoxes.Length];
        //Loops through each textBox gameObject to find their text component.
        for (int i = 0; i <= (textBoxes.Length - 1); i++) {
            myText [i] = textBoxes [i].GetComponent<Text> ();
        }
        //Loads all the high scores after the game objects used to display them have been found.
        LoadHighScores ();
    }

    void LoadHighScores () {
        //Each saved high score gets loaded into it's corresponding text box.
        myText[0].text = PersistentHighScores.bossesDefeated.ToString();
        myText[1].text = PersistentHighScores.enemiesDefeated.ToString();
        myText[2].text = PersistentHighScores.maxFloorsCleared.ToString();
        //When a new high score needs to be added, just use the same format as the lines above but inc
rease the
        //number of myText to access the new display box and reference the specific high score you wan
t to load
        //from the savedHighScores class.
    }
}
```

LevelManager

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;

public class LevelManager : MonoBehaviour {

    public float autoLoadLevelAfter;
    public int currentLoadedSceneIndex;

    private ScreenFader fader;
    private string levelName;

    //When the Awake procedure runs, levelIndex will be set to the current
    //scene's index from the build settings.
    void Awake () {
        currentLoadedSceneIndex = SceneManager.GetActiveScene().buildIndex;
        fader = GameObject.FindObjectOfType<ScreenFader>();
    }

    //If the autoLoadLevelAfter is not zero, the class will load the next level in the amount of secon
ds from
    //autoLoadLevelAfter.
    void Start () {
        if (autoLoadLevelAfter > 0) {
            Invoke("LoadNextLevel", autoLoadLevelAfter);
        }
    }

    //Loads the level with the name given to it.
    public void LoadLevel(string name){
        Debug.Log ("New Level load: " + name);
        SceneManager.LoadScene(name);
    }

    //Reloads the current level.
    public void ReLoadLevel () {
        SceneManager.LoadScene(currentLoadedSceneIndex);
    }

    //The game stops running and closes.
    public void QuitRequest(){
        Debug.Log ("Quit requested");
        Application.Quit ();
    }

    //Loads the next level in the build settings order.
    public void LoadNextLevel () {
        SceneManager.LoadScene(currentLoadedSceneIndex+1);
    }

    //Tells the fader on the level to fade out and loads the next level in the build settings order af
ter 1 second.
    public void LoadNextAfterFade () {
        fader.FadeOut();
        Invoke("LoadNextLevel", 1f);
    }

    //Tells the fader on the level to fade out and calls the LoadLevelFadeName procedure after 1 secon
d.
    public void LoadAfterFade (string name) {
        fader.FadeOut();
        levelName = name;
        Invoke("LoadLevelFadeName", 1f);
    }

    //Called by the LoadAfterFade procedure to load the level with the name assigned to the variable l
evelName from
    // LoadAfterFade. This task is split into two prodecures because 'Invoke' can't call a procedure w
ith parameters.
```

```
    void LoadLevelFadeName () {
        Debug.Log ("New Level load: " + levelName);
        SceneManager.LoadScene(levelName);
    }
}
```

OptionsMenu

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class OptionsMenu : MonoBehaviour {

    public GameObject livesDisplay, potionsDisplay, monsterHealthDisplay, startingNumberDisplay, moves
LeftDisplay;
    public GameObject menu1, menu2, menu3, buttonGrid;
    public GameObject operatorButtonPrefab;

    private LevelManager lvlManager;
    private Text lives, potions, monsterHealth, startingNumber, movesLeft, messageDisplay;
    private GenerateMathQuestion genMathQ;
    private List<int[]> answerOperations;
    private List<int> playerAnswers;
    private int[] mathsQuestion;
    private int movesLeftValue, state;
    private bool giveRewards;

    //Initialise all game objects to display data to the user and generate a maths question.
    void Start () {
        lvlManager = GameObject.FindObjectOfType<LevelManager>();
        genMathQ = GetComponent<GenerateMathQuestion>();
        messageDisplay = menu3.GetComponentInChildren<Text>();
        lives = livesDisplay.GetComponent<Text>();
        potions = potionsDisplay.GetComponent<Text>();
        monsterHealth = monsterHealthDisplay.GetComponent<Text>();
        startingNumber = startingNumberDisplay.GetComponent<Text>();
        movesLeft = movesLeftDisplay.GetComponent<Text>();
        lives.text = PersistentGameData.playerLivesSave.ToString();
        potions.text = PersistentGameData.potions.ToString();
        mathsQuestion = genMathQ.GetMathsQuestion();
        monsterHealth.text = mathsQuestion[2].ToString();
        startingNumber.text = mathsQuestion[1].ToString();
        movesLeftValue = mathsQuestion[0];
        movesLeft.text = movesLeftValue.ToString();
        answerOperations = genMathQ.GetAnswerSequence();
        answerOperations.Reverse();
        playerAnswers = new List<int>();
        CreateButtons();
        giveRewards = true;
    }

    //Check if the player has answered the question or died.
    void Update () {
        if (movesLeftValue == 0) {
            bool correct = true;
            for (int i = 0; i < playerAnswers.Count; i++) {
                if (playerAnswers[i] != i) {
                    correct = false;
                }
            }
            if (correct) {
                //Player has won battle.
                menu3.SetActive(true);
                messageDisplay.text = "You have won! You received a potion as a reward.";
                state = 1;
                Invoke("DealWithOutcome", 2);
            } else {
                //Player was not correct.
                PersistentGameData.playerLivesSave -= 1;
                lives.text = PersistentGameData.playerLivesSave.ToString();
                movesLeftValue = mathsQuestion[0];
```

```
                menu3.SetActive(true);
                messageDisplay.text = "You were not correct. The monster attacked and you lost a life.
";
                state = 2;
                Invoke("DealWithOutcome", 2);
            }
        } else if (PersistentGameData.playerLivesSave == 0) {
            //Player is dead.
            if (PersistentGameData.dungeonLevel > PersistentHighScores.maxFloorsCleared) {
                PersistentHighScores.maxFloorsCleared = PersistentGameData.dungeonLevel;
            }
            menu3.SetActive(true);
            messageDisplay.text = "You are dead. Game Over.";
            state = 3;
            Invoke("DealWithOutcome", 2);
        }
    }

    //The outcome of the player answering a question or dying.
    void DealWithOutcome () {
        switch (state) {
            case 1:
                if (giveRewards == true) {
                    PersistentGameData.potions += 1;
                    PersistentHighScores.enemiesDefeated += 1;
                    giveRewards = false;
                }
                lvlManager.LoadAfterFade("04Dungeon");
                break;
            case 2:
                playerAnswers = new List<int>();

                menu3.SetActive(false);
                menu2.SetActive(false);
                menu1.SetActive(true);
                break;
            case 3:
                SaveGameSystem.DeleteSaveGame("SavedGameData");
                PersistentGameData.saveTheGame = false;
                lvlManager.LoadAfterFade("01StartMenu");
                break;
        }
    }

    //Creates the list of buttons for the user to enter their answer.
    void CreateButtons () {
        int buttonNumber = answerOperations.Count;
        List<int> numbers = new List<int>();
        playerAnswers = new List<int>();
        for (int j = 0; j < buttonNumber; j++) {
            numbers.Add(j);
        }
        for (int i = 0; i < buttonNumber; i++) {
            GameObject operationButton = Object.Instantiate(operatorButtonPrefab, buttonGrid.transform
) as GameObject;
            Text operationButtonText = operationButton.GetComponentInChildren<Text>();
            Button operationButtonFunction = operationButton.GetComponent<Button>();
            int randomPosition = Random.Range(0, numbers.Count);
            int thisPosition = numbers[randomPosition];
            numbers.RemoveAt(randomPosition);
            string buttonText = System.Convert.ToChar(answerOperations[thisPosition][0]).ToString();
            buttonText = System.String.Concat(buttonText, answerOperations[thisPosition][1].ToString()
);
            operationButtonText.text = buttonText;
            operationButtonFunction.onClick.AddListener(delegate {WhenButtonClicked(thisPosition); });
        }
    }

    //All buttons have this function to record the player's answer.
    void WhenButtonClicked (int buttonNumber) {
        movesLeftValue -= 1;
        playerAnswers.Add(buttonNumber);
    }
```

```
    //When the player presses the heal button, one potion is used to give the player full health.
    public void Heal () {
        if (PersistentGameData.potions > 0) {
            PersistentGameData.potions -= 1;
            PersistentGameData.playerLivesSave = 3;
            lives.text = PersistentGameData.playerLivesSave.ToString();
            potions.text = PersistentGameData.potions.ToString();
        }
    }

    //Called by the fight button to switch the visible menus.
    public void Fight () {
        menu2.SetActive(true);
        menu1.SetActive(false);
    }
}
```

PersistentGameData

```
using System.Collections;
using System.Collections.Generic;
using System;
using UnityEngine;

public class PersistentGameData : MonoBehaviour {

    public static int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
    public static string playerName;
    public static Vector3 playerPosSave, exitPosSave;
    public static List<char> operatorCodesArray;
    public static int[,] dungeonMapSave;
    public static bool saveTheGame;

    //When this object is instantiated, this class will load the saved data from the file or
    //create a set of new data if there is no file and this is a new game.
    void Awake () {
        saveTheGame = true;
        bool dataLoaded = LoadGameData();
        if (!dataLoaded) {
            dungeonMapSave = new int[1,1];
            operatorCodesArray = new List<char>();
            playerPosSave = new Vector3(0,0,0);
            exitPosSave = new Vector3(0,0,0);
            dungeonLevel = 1;
            playerLivesSave = 3;
            potions = 3;
            rangeMin = 0;
            rangeMax = 0;
            maxMoves = 0;
            playerName = "";
        }
    }

    //Before this object is destroyed, save the game data to the file.
    void OnDisable () {
        if (saveTheGame == true) {
            SaveGameData();
        }
    }

    //This method is run before a new scene is loaded to save the data of the previous scene.
    void SaveGameData () {
        GameData SavedGameData = new GameData();
        SavedGameData.dungeonLevel = dungeonLevel;
        SavedGameData.rangeMax = rangeMax;
        SavedGameData.rangeMin = rangeMin;
        SavedGameData.maxMoves = maxMoves;
        SavedGameData.playerLivesSave = playerLivesSave;
        SavedGameData.playerName = playerName;
        SavedGameData.potions = potions;
        SavedGameData.operatorCodesArray = operatorCodesArray;
        SavedGameData.playerPosSaveX = playerPosSave.x;
        SavedGameData.playerPosSaveY = playerPosSave.y;
```

```
        SavedGameData.playerPosSaveZ = playerPosSave.z;
        SavedGameData.exitPosSaveX = exitPosSave.x;
        SavedGameData.exitPosSaveY = exitPosSave.y;
        SavedGameData.exitPosSaveZ = exitPosSave.z;
        SavedGameData.dungeonMapSave = dungeonMapSave;
        SaveGameSystem.SaveGame(SavedGameData, "SavedGameData");
        Debug.Log("Game data saved.");
    }

    //This method uses the SaveGameSystem to load the data from the file and returns true if it finds
the file.
    bool LoadGameData () {
        if (SaveGameSystem.DoesSaveGameExist ("SavedGameData")) {
            GameData SavedGameData = SaveGameSystem.LoadGame("SavedGameData") as GameData;
            dungeonLevel = SavedGameData.dungeonLevel;
            rangeMax = SavedGameData.rangeMax;
            rangeMin = SavedGameData.rangeMin;
            maxMoves = SavedGameData.maxMoves;
            playerLivesSave = SavedGameData.playerLivesSave;
            playerName = SavedGameData.playerName;
            potions = SavedGameData.potions;
            operatorCodesArray = SavedGameData.operatorCodesArray;
            dungeonMapSave = SavedGameData.dungeonMapSave;

            //The vector of the player's and exit's position is created out of the induvidual coordina
tes for each axis.
            playerPosSave = new Vector3(SavedGameData.playerPosSaveX, SavedGameData.playerPosSaveY, Sa
vedGameData.playerPosSaveZ);
            exitPosSave = new Vector3(SavedGameData.exitPosSaveX, SavedGameData.exitPosSaveY, SavedGam
eData.exitPosSaveZ);
            Debug.Log("Game data loaded.");
            return true;
        }
        return false;
    }
}

//This is a separate private class that inherits from SaveGame and is used by PersistentGameData to st
ore the game data
//in a serialized permentant file.
[Serializable]
class GameData : SaveGame {

    public int dungeonLevel, rangeMin, rangeMax, maxMoves, playerLivesSave, potions;
    public string playerName;
    public float playerPosSaveX, playerPosSaveY, playerPosSaveZ;
    public float exitPosSaveX, exitPosSaveY, exitPosSaveZ;
    public List<char> operatorCodesArray;
    public int[,] dungeonMapSave;

    //The vector of the player's and exit's position is saved separately as three floats of each separ
ate coordinate
    //because Vector3 is not a standard data type.
}
```

PersistentHighScores

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PersistentHighScores : MonoBehaviour {

    public static int maxFloorsCleared, enemiesDefeated, bossesDefeated;

    //When this object is instantiated, this class will load the saved data from the file or create a
set of new data
    //if there is no file and this is a new game.
    void Awake () {
        bool dataLoaded = LoadHighScoreData();
        if (!dataLoaded) {
            maxFloorsCleared = 0;
```

```
            enemiesDefeated = 0;
            bossesDefeated = 0;
        }
    }

    //Before this object is destroyed, save the game data to the file.
    void OnDisable () {
        SaveHighScoreData();
    }

    //This method is run before a new scene is loaded to save the data of the previous scene.
    void SaveHighScoreData () {
        HighScoreData SavedHighScores = new HighScoreData();
        SavedHighScores.maxFloorsCleared = maxFloorsCleared;
        SavedHighScores.enemiesDefeated = enemiesDefeated;
        SavedHighScores.bossesDefeated = bossesDefeated;
        SaveGameSystem.SaveGame(SavedHighScores, "SavedHighScores");
        Debug.Log("High scores saved.");
    }

    //This method uses the SaveGameSystem to load the high score data from the file and returns true i
f it finds the file.
    bool LoadHighScoreData () {
        if (SaveGameSystem.DoesSaveGameExist ("SavedHighScores")) {
            HighScoreData SavedHighScores = SaveGameSystem.LoadGame("SavedHighScores") as HighScoreDat
a;
            maxFloorsCleared = SavedHighScores.maxFloorsCleared;
            enemiesDefeated = SavedHighScores.enemiesDefeated;
            bossesDefeated = SavedHighScores.bossesDefeated;
            Debug.Log("High score data loaded.");
            return true;
        }
        return false;
    }
}

//This is a separate private class that inherits from SaveGame and is used by PersistentHighScores to
store the high score
//data in a serialized permentant file.
[Serializable]
class HighScoreData : SaveGame {

    public int maxFloorsCleared, enemiesDefeated, bossesDefeated;
}
```

PinchZoom

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Cinemachine;

public class PinchZoom : MonoBehaviour {

    public float orthoZoomSpeed = 0.5f;

    private CinemachineVirtualCamera playerCam;

    //Find the camera component on this game object.
    void Start () {
        playerCam = GetComponent<CinemachineVirtualCamera>();
    }

    //Every frame, the class will check whether there are two fingers touching the screen of the devic
e, it there are
    //it will do a couple of calculations to measure the difference between them and check if the user
 is trying to
    //zoom out or in and the camera size will change depending on the touch inputs.
    void Update () {
        if (Input.touchCount == 2) {
            Touch touchZero = Input.GetTouch(0);
            Touch touchOne = Input.GetTouch(1);
            Vector2 touchZeroPrevPos = touchZero.position - touchZero.deltaPosition;
```

```
            Vector2 touchOnePrevPos = touchOne.position - touchOne.deltaPosition;
            float prevTouchDeltaMag = (touchZeroPrevPos - touchOnePrevPos).magnitude;
            float touchDeltaMag = (touchZero.position - touchOne.position).magnitude;
            float deltaMagnitudeDif = prevTouchDeltaMag - touchDeltaMag;
            playerCam.m_Lens.OrthographicSize += deltaMagnitudeDif * orthoZoomSpeed;
            playerCam.m_Lens.OrthographicSize = Mathf.Max(playerCam.m_Lens.OrthographicSize, 2.5f);
        }
    }
}
```

PlayerMovement

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMovement : MonoBehaviour {

    public GameObject mainCamera;
    public float movementSpeed = 0.5f;
    public Vector2Int goal;
    public List<Vector3Int> path;

    private List<Node> openNodes, visitedNodes;
    private int[,] dungeonGrid;
    private int gridLength;
    private Camera playerCam;
    private bool characterMoving;
    private int count;
    private float tempTime;
    private LevelManager lvlManager;

    //Finds the camera, dungeon grid and gridLength.
    void Start () {
        lvlManager = FindObjectOfType<LevelManager>();
        playerCam = mainCamera.GetComponent<Camera>();
        dungeonGrid = PersistentGameData.dungeonMapSave;
        float totalElements = (float)PersistentGameData.dungeonMapSave.Length;
        gridLength = Mathf.RoundToInt(Mathf.Sqrt(totalElements)) - 1;
        characterMoving = false;
        count = 0;
        tempTime = 0f;
        gameObject.transform.position = PersistentGameData.playerPosSave;
        openNodes = new List<Node>();
        visitedNodes = new List<Node>();
        path = new List<Vector3Int>();
    }

    //Checks every frame for the player's touch on screen, if the player selects a floor tile in the d
ungeon
    //the player character will travel there. If the player is at the exit a new dungeon level is load
ed.
    void Update () {
        //If the player steps on the exit tile, the next level of the dungeon is loaded.
        if (gameObject.transform.position == PersistentGameData.exitPosSave) {
            Debug.Log("Player is at exit");
            PersistentGameData.dungeonLevel += 1;
            PersistentGameData.dungeonMapSave = new int[1,1];
            lvlManager.ReLoadLevel();
        } else if (characterMoving == true) {
            Debug.Log("Player is moving");
            tempTime += Time.deltaTime;
            if (tempTime > movementSpeed) {
                AnimateWalk ();
            }
        } else if (Input.touchCount == 1) {
            Touch fingerPos = Input.GetTouch(0);
            Vector2 tileTouchedPos = playerCam.ScreenToWorldPoint(fingerPos.position);
            goal = Vector2Int.RoundToInt(tileTouchedPos);
            if (goal.x > 0 && goal.x < gridLength && goal.y > 0 && goal.y < gridLength) {
                if (dungeonGrid[goal.x, goal.y] != 0) {
                    openNodes.Clear();
                    visitedNodes.Clear();
                    path.Clear();
```

```
                AStarPathfinding();
                characterMoving = true;
            }
        }
    //Debugging condition for searching for mouse input.
    } else if (Input.GetMouseButtonDown(0)) {
        Vector2 tileTouchedPos = playerCam.ScreenToWorldPoint(Input.mousePosition);
        goal = Vector2Int.RoundToInt(tileTouchedPos);
        if (goal.x > 0 && goal.x < gridLength && goal.y > 0 && goal.y < gridLength) {
            if (dungeonGrid[goal.x, goal.y] != 0) {
                openNodes.Clear();
                visitedNodes.Clear();
                path.Clear();
                AStarPathfinding();
                characterMoving = true;
            }
        }
    }
}

//Save the player's position before another scene is loaded.
void OnDisable () {
    PersistentGameData.playerPosSave = gameObject.transform.position;
}

//Called by the DisplayDungeon class to get the newly generated dungeon.
public void GetNewDungeon () {
    dungeonGrid = PersistentGameData.dungeonMapSave;
    float totalElements = (float)PersistentGameData.dungeonMapSave.Length;
    gridLength = Mathf.RoundToInt(Mathf.Sqrt(totalElements)) - 1;
}

//Animate the player character moving through the dungeon and give them a 1 in 18 chance of
//encountering a monster.
void AnimateWalk () {
    tempTime = 0f;
    if (count != path.Count) {
        gameObject.transform.position = path[count];
        count++;
        int chance = Random.Range(1, 19);
        if (chance == 18) {
            lvlManager.LoadNextAfterFade();
        }
    } else {
        PersistentGameData.playerPosSave = gameObject.transform.position;
        characterMoving = false;
        count = 0;
    }
}

//Pathfinds to the goal coordinates using the A* algorithm.
void AStarPathfinding () {
    Vector2Int startCoords = Vector2Int.RoundToInt(gameObject.transform.position);
    Node startNode = new Node(startCoords, null, 0, 0);
    openNodes.Add(startNode);
    Node currentNode = startNode;
    //While the path has not reached the goal.
    while (currentNode.coords != goal) {
        Vector2Int[] neighbours = FindNeighbours(currentNode.coords);
        foreach (Vector2Int n in neighbours) {
            if (n != Vector2Int.zero) {
                Node newNode = new Node(n, currentNode, 0, currentNode.gCost + 1);
                int FCost = CalculateFCost(newNode);
                newNode.fCost = FCost;
                Node duplicate = FindNodeInList(newNode, false);
                if (duplicate == null) {
                    duplicate = FindNodeInList(newNode, true);
                    if (duplicate != null && newNode.fCost < duplicate.fCost) {
                        openNodes.Add(newNode);
                    } else if (duplicate == null) {
                        openNodes.Add(newNode);
                    }
                }
            }
```

```
            }
            visitedNodes.Add(currentNode);
            openNodes.Remove(currentNode);
            currentNode.fCost = int.MaxValue;
            foreach (Node m in openNodes) {
                //Find the node with least fCost to be next current Node.
                if (m.fCost <= currentNode.fCost) {
                    currentNode = m;
                }
            }
        }
        visitedNodes.Add(currentNode);
        CreatePath();
    }

    //Creates a list of the coordinates of each tile on the path found by the A* algorithm.
    void CreatePath () {
        Node currentNode = visitedNodes[visitedNodes.Count-1];
        while (currentNode.parentNode != null) {
            Vector3Int nodeCoords = new Vector3Int(currentNode.coords.x, currentNode.coords.y, 0);
            path.Add(nodeCoords);
            currentNode = currentNode.parentNode;
        }
        path.Reverse();
    }

    //Finds a node with specific coordinates in a specific list and return it, otherwise return null.
    Node FindNodeInList (Node node, bool openList) {
        //Finds any node that has the same coordinates as the node given.
        System.Predicate<Node> duplicateNodeFinder = (Node n) => {return n.coords == node.coords; };
        //If I want to search the openNodes list.
        if (openList == true) {
            Node foundNode1 = openNodes.Find(duplicateNodeFinder);
            return foundNode1;
        //If I want to search the visitedNodes list.
        } else {
            Node foundNode2 = visitedNodes.Find(duplicateNodeFinder);
            return foundNode2;
        }
    }

    //Calculates the fCost of a node using the diagonal distance heuristic.
    int CalculateFCost (Node node) {
        int xCoord = node.coords.x;
        int yCoord = node.coords.y;
        int hCost = Mathf.Max(Mathf.Abs(xCoord-goal.x), Mathf.Abs(yCoord-goal.y));
        int fCost = node.gCost + hCost;
        return fCost;
    }

    //Find the given tile's neighbours.
    Vector2Int[] FindNeighbours (Vector2Int coords) {
        int xCoord = coords.x;
        int yCoord = coords.y;
        Vector2Int[] NESWArray = new Vector2Int[4];
        NESWArray [0] = new Vector2Int (xCoord, yCoord + 1);
        NESWArray [1] = new Vector2Int (xCoord + 1, yCoord);
        NESWArray [2] = new Vector2Int (xCoord, yCoord - 1);
        NESWArray [3] = new Vector2Int (xCoord - 1, yCoord);
        //Get rid of coordinates that are walls or outside the grid.
        for (int i = 0; i <= 3; i++) {
            if (NESWArray [i].x > gridLength || NESWArray [i].x < 0) {
                NESWArray [i] = Vector2Int.zero;
            } else if (NESWArray [i].y > gridLength || NESWArray [i].y < 0) {
                NESWArray[i] = Vector2Int.zero;
            } else if (dungeonGrid[NESWArray[i].x, NESWArray[i].y] == 0) {
                NESWArray[i] = Vector2Int.zero;
            }
        }
        return NESWArray;
    }

}
```

```
//The class used to create nodes, which keeps a Node's data in one object.
class Node {

    public Vector2Int coords;
    public Node parentNode;
    public int fCost, gCost;

    //A node keeps track of it's coordinates, it's parent node and it's f and g costs.
    public Node (Vector2Int coOrds, Node parent, int costF, int costG) {
        coords = coOrds;
        parentNode = parent;
        fCost = costF;
        gCost = costG;
    }
}
```

SaveGame

```
using System;

[Serializable]
public abstract class SaveGame {
    // Save game template.
}
```

SaveGameSystem

```
using UnityEngine;
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

public static class SaveGameSystem {

    //Takes a saveGame object and saves it to the file name - 'name'.
    public static bool SaveGame(SaveGame saveGame, string name) {
        //Create a Binary formatter.
        BinaryFormatter formatter = new BinaryFormatter();
        //Create the file stream to where the game will be saved if the file does not already exist
        //a new file will be created otherwise it will overwrite the previous file.
        using (FileStream stream = new FileStream(GetSavePath(name), FileMode.Create)) {
            try {
                //Serialise the saveGame object using the binary formatter.
                formatter.Serialize(stream, saveGame);
            }
            catch (Exception) {
                //Returns false if the saveGame object cannot be saved for any reason.
                return false;
            }
        }
        //Returns true if the saveGame object is saved.
        return true;
    }

    //Deserializes the saveGame object with the file name - 'name'.
    public static SaveGame LoadGame(string name) {
        if (!DoesSaveGameExist(name)) {
            //If there is no save with the name - 'name', load nothing.
            return null;
        }
        //Create a binary formatter.
        BinaryFormatter formatter = new BinaryFormatter();
        //Open the file stream to the saveGame file.
        using (FileStream stream = new FileStream(GetSavePath(name), FileMode.Open)) {
            try {
                //Deserialize the file using the binary formatter and return it as a saveGame object.
                return formatter.Deserialize(stream) as SaveGame;
```

```
            }
            catch (Exception) {
                //Returns false if the saveGame object cannot be loaded for any reason.
                return null;
            }
        }
    }

    //Deletes a saveGame file with the file name - 'name'.
    public static bool DeleteSaveGame(string name) {
        try {
            //Deletes the file.
            File.Delete(GetSavePath(name));
        }
        catch (Exception) {
            //Returns false if the file cannot be deleted for any reason.
            return false;
        }
        //Returns true if the file is deleted.
        return true;
    }

    //Tries to find the save file with the name - 'name'.
    public static bool DoesSaveGameExist(string name) {
        //Returns true if the file exists otherwise returns false.
        return File.Exists(GetSavePath(name));
    }

    //Finds the file path to the file with the name - 'name'.
    private static string GetSavePath(string name) {
        //Returns a string containing the complete path to the file.
        return Path.Combine(Application.persistentDataPath, name + ".sav");
    }

}
```

ScreenFader

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ScreenFader : MonoBehaviour {

    private Animator anim;

    //Initializes the animation component of the fader game object.
    void Start () {
        anim = GetComponent<Animator>();
    }

    // Calls the animator to trigger the fade in sequence of animations.
    public void FadeIn () {
        anim.SetTrigger("FadeInTrigger");
    }

    // Calls the animator to trigger the fade out sequence of animations.
    public void FadeOut () {
        anim.SetTrigger("FadeOutTrigger");
    }
}
```