This document contains evidence for the development of the second prototype. This will include the iterative development of the code as well as notations of any errors/corrections and testing done during the development of the prototype.

**Setting Up Unity – Current Status**Most the folder and scene organisation was done during the start of the first prototype development. Starting this prototype however will come with some minor adjustments to the folders. I have added an additional 3 folders to the project: Objects, Prefabs & Miscellaneous. As I am now working with player objects and entities, Objects and Prefabs will be needed to organise them. Miscellaneous is not something that is required, it is more so added for the sake of ease. Anything that I build that may not be used or doesn’t match up with any other folder will most probably end up in this folder if it doesn’t get deleted.

**Code & Object Removal**In the first prototype, there were two scripts that were included. *‘ExitMapGeneratorScript’* was built as a method to exit the procedurally generated map and go back to the main menu via a button on a canvas object – a way to easily perform multiple tests without constantly closing the Unity editor or built program. *‘OrbitalCamera’* was a script that was made to have a way of viewing the procedurally generated map from all angles using a camera. These are now not required on this prototype and as such are deleted.

**Project Development – PlayerScript & Player Object**The main component of this prototype is the player object and the player script. The main form of control for the player and will provide me with the ability to see the map from the main perspective view which will give the chance to check through the map again for areas such as the height of walls and overall size of the map.

Code Development – PlayerScript  
The algorithm for the movement and functionality of the player will be included into one script. At this point in time the mechanics of the player object that are being implemented are just the movement and the camera control.  
The script makes use of four main private float values which are essential to its function: *‘movementSpeed’* is the base set value for the walking speed of the player object; *‘mouseSensitivity’* is the set value for mouse sensitivity (though this value is overwritten with raw mouse input); *‘verticalRotation’* is an initialisation value this is used to calculate the y-axis rotation of the camera; and the final value *‘visionTaxisRange’* works as the angle range for the y-axis rotation of the camera.

The rotation algorithm in the *‘PlayerScript’* makes use of a library known as Quaternion; this library works with rotations and contains multiple calculations that you can do for rotations. The one used here is Euler, which perform rotation for z, x and y-axis around their respective axis based on provided values.

Unity Development – Player Object  
This script that was written will be implemented into an object in Unity which will act as the game’s player object. The object itself is a simple 3D capsule object – a curve cylindrical shape. As the game works in the first-person perspective, it will seem slightly unnecessary to have the object as a detailed entity. Fortunately, when added into the environment, the capsule object already comes with a collider mesh and physics. The most important part of the capsule object is the capsule collider mesh which will be interacting with everything else in the game; because of this, the capsule itself is actually embedded into the plane – the collider mesh works with calculating collisions but the object itself works with the physics and gravity of the entire object, embedding it into the ground keeps it on the plane but removes it from view.

While programming the player object, I realised something that I could have done to combat the visual entity of the capsule – transparency. With most major video games that are a first-person perspective, to save rendering costs the player is not rendered at all; control and functionality is still there but visually, it doesn’t exist. This was done was done via a short line of code that simply disabled rendering.

Code Development – Map Collisions  
As you can see from the two images above showing gameplay, I haven’t added collisions into my procedurally generated map. The player is easily able to pass through all of the walls on the map, eventually being able to run off the plane and fall out of the map. To add collisions into the walls and prevent the player from passing through, I need to go back into my *‘MeshBuilder’* script. I created a new mesh which is essentially a duplicate of the wall mesh, this mesh has included collision physics components – this makes the walls “solid” and in turn, prevent the player from clipping through the walls.

Error Review – Walls Not Rendering  
The walls being made via the *‘MeshBuilder’* script work on an algorithm known as marching squares – where it builds triangles based on 4 points of a square. This was written in the first prototype with the initial building of the script. I haven’t realised until this moment that some of the cases for the algorithm were in correct, this means that some of the walls were not being built allowing for the player to be trapped inside the walls when collisions were added. I went through the marching squares cases and corrected the issues.

Code Development – Player Spawning  
With the player object made, I can start working on the player spawning code. This code was initially designed to be a completely new script embedded into the generated map object, but after looking through my code, incorporating it into an already existing script as another function will be easier and cleaner for continued development.  
The algorithm of the code requires it to search the array made for the procedural map generation to find a position where the map is a floor (this where the value of map[x, y] would equal to 0) and spawn a clone of the player object at that position.

Error Review – Incorrect Spawning  
When the code was implemented, the player could spawn in the scene using the position given from the array used to build the procedurally generated map. The information was taken and compiled successfully but on multiple generations, the player object spawned away from the map and in turn fell into the endless void of the unity scene – it would also on multiple occasions spawn the player within a wall (luckily it doesn’t render the player ‘stuck’.) I didn’t know at the time what was causing this and I tried multiple methods, including creating a new array specifically for the player spawn positioning.

The solution to at least to half of the problem was through the values used from the array to the measurements in the unity scene; comparing the distancing in the array to the unity scene, the ratio of distancing is 2:1 meaning that the value 50 in the code and array was actually 25 in the unity scene itself. In order to correct this I simply needed to divide the x-axis and y-axis values by 2 after they were taken from the array. This now means the player will now always spawn within the map – there is still an issue with the player spawning into the walls of the maps generated.

The issue with the spawning in the walls is also due to the values in the array compared to the measurements in Unity. The positions in Unity work similarly to co-ordinates, with (0,0,0) being the absolute centre of the map with negative values being towards the left and positive values being towards the right; the array of course doesn’t work like this – if you were to place the array on top of the floor of the map, positon [0,0] on the array would be [-50, 50]. The solution I came up with was to first multiply the y-axis value by -1 before adding the value 50. This will set up the positions in the Unity environment correctly: the value of 100 in the array would be 50 in the Unity environment which is what it is supposed to be.

Error Review – Clipping In-between Walls  
While the mathematics of the spawning were corrected, there was still an issue with the player object spawning in the walls. This was actually due to the map modifications. Within the code, there is a function that changes the walls to floors and vice versa if the floors/walls were of ‘inadequate’ value. I performed a quick fix by simply disabling the ability for the floor tiles to be converted into walls. This removed the problem of tight corner walls and thus removed most case that the player object would spawn into a wall.

Code Development – Centre Spawning Hub  
Even with the changes I made to the procedurally generated map, I still had continued issues with the player spawning. Up until this point I have not been able to solves these issues and I have decided to create an alternative spawning method; within the map generation code, I have setup a specific point in the centre to be a room – this provides a set place where the player can spawn and while it is not the procedural spawning I initially wanted, this is an acceptable alternative.

Final In-Development Testing  
The centre spawning hub concludes the development of the second prototype; with a lot of changes and modifications made, I want to perform an In-Development test before I perform a testing and evaluation with the peer group audience. Here I want to make sure that there are no issues with spawning the player in the procedurally generated map and that the generation still creates a fully connected map even with the considerable modifications made.